MPW C++ 3.1 / MPW 3.2 Compatibility Note

The software in the MPW C++ 3.1 package was initially tested with version 3.1 of MPW C and the MPW Development Environment. Now it has also been tested with version 3.2 of MPW C and the MPW Development Environment. The main MPW C++ 3.1 software is compatible with both versions of MPW C and MPW. However, the MPW C++ 3.1 *examples* require some minor changes (described below) when using version 3.2 of MPW C and MPW.

Please note, as long as you have a copy of MPW C 3.2, we recommend using the MPW C 3.2 compiler rather than the MPW 3.2b1 compiler included with the MPW C++ 3.1 package.

Examples and Load/Dump

There is an error in the examples that is visible when using the load/dump feature of MPW C++. To fix this problem, find the classes derived from "HandleObject" and specify that they are "public HandleObject". For example, in "Shapes.h"

```
class TShape : HandleObject {
     // ...
};
```

should be changed to read as

```
class TShape : public HandleObject {
     // ...
};
```

HandleObject has its own "operator new" that allocates by handles rather than pointers. This operator is hidden if you do a private derivation.

Changes to Make Files for the MPW C++ Examples

Count

When using MPW 3.2, make the following modifications to the ObjectFiles list of "Count.make":

```
"{CLibraries}"CRuntime.o becomes "{Libraries}"Runtime.o
"{Libraries}"Interface.o is deleted
```

Thus the definition of ObjectFiles in the modified makefile should look like:

```
ObjectFiles     =     "{Libraries}"Stubs.o ∂
                      "{Libraries}"Runtime.o ∂
                      "{Libraries}"Interface.o ∂
                      "{CLibraries}"CPlusLib.o ∂
                      "{CLibraries}"StdCLib.o ∂
                      "{Libraries}"ToolLibs.o ∂
                      Count.cp.o ∂
                      StreamCounter.cp.o
```

CPlusShapesApp

Make the same changes to the makefile "CPlusShapesApp.make" under the rule for building CPlusShapesApp (starts on line 87). The new build rule will look like:

```
CPlusShapesApp ƒƒ {Objs} CPlusAppLib.o CPlusShapesApp.make
     Link -d -mf -o {Targ} {SymOpts} ∂
          {Objs} ∂
          CPlusAppLib.o ∂
          "{CLibraries}"CPlusLib.o ∂
          "{Libraries}"Runtime.o ∂
          "{CLibraries}"StdCLib.o ∂
          "{Libraries}"Interface.o
     SetFile {Targ} -t APPL -c 'MOOT' -a B
```

TESample

The same library changes must be made to "CPlusTESample.make" under the rule for building CPlusTESample (starts on line 96). The new build rule should look like:

```
CPlusTESample ƒƒ {Objs} CPlusAppLib.o CPlusTESample.make
     Link -d -o {Targ} {SymOpts} ∂
          {Objs} ∂
          CPlusAppLib.o ∂
          "{CLibraries}"CPlusLib.o ∂
          "{Libraries}"Runtime.o ∂
          "{CLibraries}"StdCLib.o ∂
          "{Libraries}"Interface.o
     SetFile {Targ} -t APPL -c 'MOOT' -a B
```

In addition, add "-mf" to the C++ compile options by changing the definition of CPlusOptions on line 94 to be:

```
CPlusOptions = {SymOpts} -mf
```

MPW C++ 3.1 Release Notes

6 September 1990

Overview

This release is Apple's first "final" release of *MPW C++*. It bears the number 3.1 because it is intended for use with *MPW 3.1*. (It is also compatible with *MPW 3.2 beta 1*, and we intend that it will also be compatible with *MPW 3.2 final* when that becomes available.) The CFront program included is called version 1.0. It is based on the final release of AT&T CFront 2.0.

New

This release of MPW C++ includes several new features (see below):

- Load/Dump (for precompiled header files)
- MultiFinder Memory Option
- MPW Shell "Mark" generation

Reporting bugs

If you find a **bug**, please report it to Apple. Use the application "Outside Bug Reporter," found on one of the release disks. After completing the bug report, send it to:

via AppleLink: CPLUS.BUGS, TRISH, and MACDTS
(please send it to ALL 3 of the above)

or

via Internet: cplus.bugs@apple.com@internet#

or

via U.S. Mail:

Apple Computer, Inc.
Developer Technical Support MS 75-3T
20525 Mariani Ave.
Cupertino, CA 95014

System Requirements

MPW 3.1 requires a hard disk and at least 2 Mb of RAM. Actually, the MPW Shell's MultiFinder partition size comes set at 1024k. On a 1 Mb Macintosh, however, the MPW Shell only gets about 800k (not using MultiFinder), and only trivial C++ programs will successfully compile. In order to compile and link large C++ programs or programs with symbolic information, it is necessary to increase MPW Shell's memory partition size to 2048k or more or use the new '-mf' option. (To change the memory partition size, simply select the MPW Shell icon in the Finder and use the "Get Info" command from the File menu. Type in a memory partition size in the "Application Memory Size" box.) SADE requires MultiFinder and at least 2.5 Mb of RAM. MPW 3.1 requires System 6.0.2 (or later) and Finder 6.1 (or later). To run SADE and the MPW Shell to debug an MPW Tool requires at least 4.5 Mb of RAM.

Note: There are situations where the MPW C compiler runs out of stack space and crashes or corrupts the heap, when compiling code emitted by CFront. Increasing the MPW Shell's stack size to 64K seems to cure these problems. (To change the MPW Shell's stack size, use the application ResEdit to open the 'HEXA' #128 resource. This resource contains a long-word which is used to set the stack size, (e.q. the value $00010000 would set the stack size to 64K). A value of zero instructs the Shell to use its default size. NOTE: Do not set the stack size to be less than about 32K manually -- for sufficiently small values the Shell won't even be able to boot without running out of stack space.) If you run across any examples in which CFront itself runs out of stack space, please let us know.

Installation

To install from diskettes:
Start up the MPW Shell.
Insert the diskette marked **'MPW C++ Disk1'**.
Type

```
Duplicate 'MPW C++ Disk1:Install' :
Install
```

Fans of the Backup command can also use the following commands:

```
Backup -from 'MPW C++ Disk1:' -to "{MPW}" -r -a -c
Backup -from 'MPW C++ Disk2:' -to "{MPW}" -r -a -c
```

NOTE: The new C compiler will be placed in **"{MPW}"Tools:C3.2b1:C**, not in **"{MPW}"Tools:C**. We did this so that you can save away your old C compiler before installing the new one. (Just move it into the **:Tools:** directory.)

NOTE: The file CPlus.help on the disk **MPW C++ Disk1** contains help information for the CPlus and CFront tools in addition to the information contained in the **MPW.Help** file. To access this additional information, type

```
Help -f {MPW}CPlus.help CPlus
```

or simply paste the information in the CPlus.help file into your MPW.help file in the appropriate places.

Parts List

The following is a listing of the components of the *MPW C++ 3.1* release. All of the pieces in the following parts list are **required** for using the *MPW C++ 3.1* release, you cannot "mix and match" with parts from previous releases:

:Tools:

CFront
C (3.2b1)
Unmangle

:Scripts:

CPlus

:Libraries:CLibraries:

CPlusLib.o/CPlusLib881.o
CPlusOldStreams.o/CPlusOldStreams881.o

:Interfaces:CIncludes:

fstream.h
generic.h
iomanip.h
iostream.h
new.h
oldstream.h
stdiostream.h
stream.h
strstream.h

:Examples:CPlusExamples:

Count.cp
Count.make
CPlusShapesApp.make
CPlusTESample.make
Instructions
Shapes.cp
Shapes.h
ShapesApp.cp

ShapesApp.h
ShapesApp.r
ShapesAppCommon.h
ShapesDocument.cp
ShapesDocument.h
StreamCounter.cp
StreamCounter.h
TApplication.cp
TApplication.h
TApplication.r
TApplicationCommon.h
TDocument.cp
TDocument.h
TECommon.h
TEDocument.cp
TEDocument.h
TESample.cp
TESample.h
TESample.r
TESampleGlue.a

Miscellaneous:

Macsbug 6.1 Unmangler
unmangle.o
CPlus.help
C++ 3.1 Release Notes *(this document)*

The C Compiler Included in This Release (3.2b1)

The *MPW 3.1* C compiler does NOT support load/dump. In order to release this feature, it was necessary for us to release a newer C compiler. It is called *3.2b1* and is needed only until *MPW 3.1* is released.

This C compiler supports some optimizations not available with MPW 3.1 C. There is good news and bad news here: the good news is that this C

compiler generates better code. The bad news is that it is not as "mature" as MPW 3.1 C and it may have some strange and wonderful bugs. If you think this compiler is generating bad code, please try the "-opt off" option on the CPlus command line. This turns off all optimizations in the C compiler.

If you *really* have trouble with the *3.2b1* compiler, please notify us about it and go back to using the *3.1* C compiler. (You won't be able to use load/dump, but at least you'll have a reliable compiler.)

There are two new C options that CFront passes through to the C compiler:

```
-opt on|off          # turn optimization on (default) or off
```

-opt off is safer than -opt on (the default). -opt on will give you faster, smaller code, but the C compiler will take longer to execute with -opt on, and it may generate incorrect code in some cases. If you write code that behaves differently depending on whether optimization is turned off or on, please notify us. This is never supposed to happen.

```
-trace on          # generate tracing code--calls to %_EP and %_BP
-trace off         # generate no tracing code (default)
-trace always      # always generate tracing code--calls to %_EP and %_BP
-trace never       # never generate tracing code (default)
```

The -trace option is most frequently used with the MacApp debugger. -trace on generates a call to %_BP at the start of each function and a call to %_EP at the end of each function. If you compile and link with -trace on, but not within MacApp, you will probably get link errors: the linker will not be able to find the %_BP and %_EP functions unless you have written your own.

Load/Dump Option

The load/dump facility can greatly reduce compile times when large header files are used. The portion of the compilation that reads the header files can be eliminated except for a few seconds to read a large disk file.

The "-dump <filename>" option saves the state of the C++ compilation (mostly the compiler symbol table) to the specified file. A matching "-load" restores the state. In effect, the compilation is suspended and resumed—the dump portion of the compile is only done once, while the load portion can be repeated for many different source files.

For example, you could say:

```
CPlus header_file.h -dump header_file.h.dump
CPlus source_one.c  -load header_file.h.dump
CPlus source_two.c  -load header_file.h.dump
```

If the dump file name ends with ':', it is suffixed with the source file name and ".dump", so the first command is equivalent to

```
CPlus header_file.h -dump :
```

The "-load" compilation can include additional headers if needed, but only one load file can read during a compile. The load occurs before any "-d" or "-u" options, so macros can be added or removed on the command line.

If you have global definitions in the header file, you will get a nontrivial "header_file.h.o" file, which you may want to include in your link. In particular, this may happen because of nontrivial "const" declarations such as "const char string[] = "abc";".

Static objects in a header file will be compiled into the "header_file.h.o" file, where they will be inaccessible from other compiles, even one that uses "-load". Because of this, static objects cause a compilation warning.

If your headers are protected with the "#ifndef...#define...#endif" convention, you can leave your #include's in the source file since the #define symbol will be defined in the load. But you should put a #ifndef...#endif around the line containing the #include itself in the source file, so that you don't waste time reading the header files just to skip them.

To generate a compressed dump file, use "-dumpc" ("-load" is used for both compressed and uncompressed versions). This reduces the file size by about 40%, with very little change in compile speed on the Mac II or Mac SE. On machines where the CPU-to-disk speed ratio is higher than the Mac II, compressed loads may actually run faster than uncompressed. In future releases, all dumps may be compressed.

The dump file contains the state of the "-mc68881", "-elems881", "-n", "-x", and "-sym" options given when the dump file was made. When loading from this dump file, the options on the command line must match those in the dump file. If they do not, the compiler will issue a warning and exit.

The compiler will also issue a warning if -d and -u are used with -load. This is not an error, but most of the time it is not what the user intended to do: typically the user expects -d and -u to affect every file included in his program. With -load they will NOT affect the code included in the dump file. The macro values in a dump file are "frozen" at the time that dump file is made.

It is legal to have *both* the -dump and -load options on the same line. You might want to do this, for example, to load from a dump file containing all the MacApp headers, compile your own headers, and then dump the whole thing into one big dump file.

Load/Dump requires a special version of the C compiler (*MPW C 3.2b1*), included in this release.

Why did Apple add Load/Dump to C++?

The Load/Dump mechanism built into *MPW C++* is designed to counteract a common problem in large scale application development and in OOP, in general. The problem is that in order to compile one small source file (typically < 20K) the compiler must first process a much larger body of header file information (22,000 lines or 725K bytes for the MacApp and Toolbox headers). Since the information in the header files rarely changes, much of this overhead can be eliminated. Load/Dump allows you to "pre-compile" the header files and then rapidly load them into the compiler at startup. Typical speed improvements are on the order of 2–3x.

Step by Step Instructions

1. Decide which header files you want to pre-compile. Typically this will include your Application Framework (MacApp, etc.), Toolbox Interface, and Language Library header files.

2. Create a new header file called MyAppDump.h which will contain #include directives for all of header files you decided to pre-compile in step 1.

   Look through your application's source and header files for #include references to any of the files from step 1. In each application file that contains a reference, paste a copy of the #include lines into MyAppDump.h and replace the #include lines in the source file with the following:

   ```
   #ifndef __MyAppDump__
   #include "MyAppDump.h"
   #endif
   ```

3. Bracket the body of MyAppDump.h with a #ifndef...#endif so that it looks like:

```
#ifndef __MyAppDump__
#define __MyAppDump__
...
#include <Quickdraw.h> // to be precompiled...
...
#endif
```

4. Add a build rule to your makefile corresponding to the compile and dump of the precompiled headers:

```
MyAppDump.h.o ƒ MyAppDump.h   etc.
  CPlus MyAppDump.h -dumpc MyAppDump.h.dump ∂
    -o MyAppDump.h.o {CPlusOptions}
```

5. Modify the build rules of your application's source files to include a "-load" of the precompiled headers, and a dependency on the dump file itself:

```
MySrc.cp.o ƒ MySrc.cp MySrc.h MyAppDump.h.o
  CPlus MySrc.cp -load MyAppDump.h.dump ∂
    -o MySrc.cp.o {CPlusOptions}
```

   (well, actually the .o corresponding to the dump — Make doesn't yet handle dependencies for tools producing multiple output files)

6. Finally, add the object file produced by the dump phase (here MyAppDump.h.o) to your link dependency and link list since it may contain definitions of global const objects, vtables, bodies of inline virtual functions, etc.

Before You Report a Load/Dump Bug...

Because of the way this release is packaged, a lot of you might see the C compiler crash in the following way when you try load/dump:

```
CPlus "{CIncludes}"stdio.h -dump stdio.dump
# C - Fatal Error : 430
# unexpected token in the token input file
#-------------------------------------------------------------
     File "HD:MPW:Interfaces:CIncludes:stdio.h"; Find •!0:§!1;
Open "{Target}"
#-------------------------------------------------------------
# C - Aborted !
```

This is probably because you are using an old C compiler that does not support load/dump. The C compiler sees the tokens that CFront sends it for load/dump and does not understand them. Get the C compiler installed in **"{MPW}"Tools:C3.2b1:C** and put it in **"{MPW}"Tools:C**. The 3.2b1 C compiler understands the load/dump tokens.

The above example is also a good test for load/dump. If you cannot make a dump file out of stdio.h, then you have something very wrong with your system.

Multifinder Memory Option

When the "-mf" option is used, MPW C++ will request MultiFinder memory when the MPW shell partition is not large enough to complete the compilation. Of course, the amount of MultiFinder memory available depends on what other applications are running; and C++ use of this memory will reduce the amount of memory available for running other applications.

MPW C++ releases MultiFinder memory at the end of each compilation, even when interrupted with Command-period. However, after certain catastrophic errors (such as a bus error recovered via "g stoptool" in Macsbug), MultiFinder memory may not be released. If this occurs, you

must quit the MPW shell to free the MultiFinder memory.

With the -mf option, you can keep the MPW shell partition small enough (e.g. 1-2 Mb) to run other applications with MPW, and still be able to do large compilations without sizing up the MPW shell.

The MPW Linker ("Link") and Lib tool also accept the -mf option and we recommend you use it whenever possible.

The CPlusShapesApp example has been modified to use -mf. The CPlusTESample example has not. You can compare the two to see the difference.

Generating Marks

The "-mark" option tells MPW C++ to install MPW shell "marks" into the source files(s). Note that only the file(s) listed on the CPlus command line are marked; include files are unaffected. (Include files are read so often that marking them each time would significantly degrade performance.)

Include files can be marked by compiling them directly:

```
CPlus -mark all something.h
```

You can mark functions, types (including classes), and global data items. The "-mark" option is followed by a list of one or more sub-options [fcts,types,data,all]; the syntax is similar to the -sym option.

The added C++ marks begin with "—" (option-underscore). Any existing marks beginning with "—" are removed, and the new marks are merged with the old non-C++ marks. The marks you added are thereby preserved (unless they start with "—").

Since the marks are maintained in the resource fork of the source file, there is no danger of corrupting the text. The modification time of the

file is not changed by the -mark option. (Actually, it's reset to its original value.) Marking does not work with source files containing "#line" directives.

Pragmas

Unrecognized pragmas (for now, all but "#pragma segment") are now passed through to the C compiler. However, any pragmas encountered within a function body, class body or other file-level definition are emitted before the definition that contains them. Since there is not an exact one-to-one mapping of C++ constructs to C constructs, position-dependent pragmas may not produce the expected effect. In particular, C type declarations used to implement C++ types may be emitted out of order or not emitted at all, if not needed.

Unmangling Tools

The **Unmangle** tool is an *MPW* tool which attempts to "unmangle" the name passed to it on the command line. (CFront appends a "type signature" to the names defined by the user; such a name is called a "mangled" name. The type signature is necessary in order to distinguish different overloaded functions. The linker cannot understand overloading, so CFront resolves it by making distinct names the same way a user would in a language like C that does not allow overloading.)

The command

```
unmangle mangled_name
```

yields the unmangled name, or reports that the name was not a mangled one. This can be useful when deciphering error messages from the linker about unreferenced externals with mangled names.

NOTE: This tool, and the library described below, are not identical to the AT&T unmangler. For example the unmangle tool and library do not unmangle local variable names.

The file "Macsbug 6.1 Unmangler" on the disk "MPW C++ Disk2" contains a resource that should be pasted into your MacsBug 6.1 "Debugger Prefs" file with ResEdit. This resource allows MacsBug 6.1 to unmangle the function names produced by C++. Thus, instead of seeing mysterious, and potentially unsettling, names like

```
__ct__8TPN_ExprFPcN21
```

in the disassembled code, or in the "pop-up name selector window" (type command-colon) you'll see

```
TPN_Expr::TPN_Expr(char*,char*,char*)
```

instead. CFront/C generates these MacsBug names by default or with the command line option "-mbg full".

(A translation by hand of the above name: "__ct" means "constructor". "__8TPN_Expr" means this is a member function of "TPN_Expr". 8 is the number of characters in "TPN_Expr". "F" means this is a function as opposed to a static data field or some other kind of object. "Pc" means "pointer to char". "N21" means 2 parameters of the same type as the 1st parameter, i.e. repeat "char *" twice. The unmanglers do this translation for you. If you need a detailed explanation of this encoding, see Bjarne Stroustrup's paper, *Type-Safe Linkage for C++*.)

We have also included the unmangler in library form in the file unmangle.o. The function unmangle(), which decodes C++ mangled symbols (i.e. a symbol with a type signature), behaves in the following manner:

```
int unmangle(char *dest, char *src, int count);
```

where:

dest = pointer to result buffer
src = pointer to mangled symbol
count = max chars to write not including null
(e.g. sizeof(buffer) - 1)

The return codes are:

-1 = error; probably because symbol was not mangled, but looked like it was
0 = symbol wasn't mangled; not copied either
1 = symbol was mangled; unmangled result fit in buffer
2 = symbol was mangled; unmangled result truncated to fit in buffer (null written)

Grunge-Level Details

Here are some interesting "bits-o-trivia" you may need to know:

• Some C++ users may be curious about where and when CFront chooses to emit virtual tables. Normally you don't have to worry about virtual tables (*vtables*), which are generated to support virtual functions. MPW C++ usually generates the tables in just one object file (out of all the files that use the class), and the link will work correctly. There should be no duplicate or missing vtables. (Vtable names begin with "__ptbl" and "__vtbl".)

You should be careful to recompile all source files which reference a class when you change the class, even if you think the change does not affect some files. If you don't recompile the file that happens to generate the vtables for the class, the tables will not be updated and your program may not run correctly.

Sometimes the vtables for a class are generated from all source files that use the class. When this happens, you will see a "duplicate symbol"

warning from link. You can safely ignore this warning. You can also fine-tune the vtable generation using the "-vtbl0" and "-vtbl1" options. For example, you can create a file which includes all of your class declarations, then compile that file with "-vtbl1" and all others with "-vtbl0".

In some rare cases, e.g. when a virtual function is written in assembly code, you may need to know how CFront decides which file to put the vtables into. The exact criteria are subject to change, but the current method is given below. (This description is only approximate.)

In general, CFront tries to find a "key" function in the class. For regular classes (not derived from PascalObject), this function is the first non-inline virtual function in the class definition. (A function is considered inline if the body is given in the class, or if the definition of the function uses the "inline" keyword.) If there is a key function, then virtual tables will be generated for a file only if that function is defined (has a body) in the file. Since only one file should define the key function, only one copy of the vtables should be produced. If there is no key function, vtables are always generated. This may lead to duplicate vtables and warnings from the linker.

Classes derived from PascalObject use a similar method to determine when to emit Pascal method tables. However, the criteria for selecting the key function are too complex to describe here.

• In order to handle static constructors and destructors in C++ code, CFront and the Linker conspire to create a special segment named "%_Static_Constructor_Destructor_Pointers". This segment is checked as the application starts up to see if any static data initialization/cleanup is required. Removing this segment or changing its name will cause this process to fail.

Recent Changes and Bug Fixes

The following is a summary of the bugs fixed since the B1 release of *MPW C++* (based on AT&T 2.0 Final). First time users of *MPW C++* will also be interested in this list since most of these bugs are present in the AT&T 2.0 Final software, but have been fixed in subsequent versions of Apple's *MPW C++*.

Bugs fixed in the final release:

• Overloaded operators in nested classes caused the load/dump mechanism to get a bus-error.

Bugs fixed in the B4 release:

• The code generated to access members of virtual base classes is now correct. This "known bug" had been so bad in previous versions that it was unsafe to use virtual base classes at all. It should now be safe.

• The name of the compilation unit was given to C and SADE incorrectly and resulted in bizarre warnings from C -- for example, "Can't find modification date on file." The name given was garbage due to an uninitialized data structure.

• "Volatile" was removed from the grammar. It is still a keyword. Therefore it is not legal to define an identifier called "volatile" and anywhere you use the keyword "volatile", you now get a syntax error. Formerly, CFront would silently generate incorrect C code for it. "Volatile" is not implemented in the 3.1 B4 release and until it is it will not be legal in the grammar. This way users will not rely on it when it does not do what they expect it to.

Bugs fixed in the B2/B3 releases:

• Although the C++ Reference manual states (p. 3) that all identifiers

containing a double underscore are reserved, such names are widely used (including in the C++ MacApp headers). Although CFront accepts these names, they can cause problems ranging from bad code to compiler bus errors. As a partial solution to this problem, MPW C++ version B1 emitted warnings when identifiers most likely to cause problems were used.

In this release such identifiers are handled correctly. However, double underscores should still be used sparingly, since you may inadvertently write an identifier which duplicates a name invented by CFront (for example, temporary variable names). If this happens, you will get a C compilation error because of the duplicate identifier.

• Referencing a type name as if it were a member function could cause a compile-time bus error.

• Initializing a local static structure with another structure caused a compile-time bus error.

• Incorrect virtual tables were generated when a visibility declaration was used. This could cause a run-time bus error.

• In a void function, a return statement with a return value of type void (as opposed to no return value) was not flagged with a warning.

• Comparison of a "pointer to member function" variable to a constant "pointer to member function" generated bad intermediate C code, causing the C compiler to emit errors.

• Comparison of two "pointers to member functions" using the "!=" operator caused incorrect C code, so result of the compare was sometimes wrong.

• When using an overloaded "operator new" with extra arguments, at a point where memory is nearly exhausted: if a "new" using extra arguments failed, and a regular (no extra arguments) version was provided, the regular version was called; this was wrong. For example, if

the first call failed and the second call succeeded (because it allocated less memory), the object would be smaller than expected.

• Extra commas in the middle or at the end of an enumerator list (in an "enum" declaration) were not flagged as a syntax error. (One extra comma at the end is allowed.)

• Operator conversion functions could not be pure virtual.

• Default arguments in the (base/member) initialization list of a constructor sometimes caused a bogus "sorry, not implemented" error.

• When a constructor used a pointer declared to point to an abstract class (but really pointing to a derived class), and called a member function which was pure virtual in the abstract class, a bogus compilation error was emitted. The error should only be emitted when the pointer is "this", since calls using "this" in a constructor are non-virtual.

• When a pointer of the form "&ptr" (where "ptr" was a class object) was cast to point to a virtual base class, CFront aborted with an internal error.

• A bogus warning was emitted for certain "switch" statement where the switch expression was an enumeration. CFront warned about possible missing cases when there were almost, but not quite, as many cases as enumeration values. This warning should not have been emitted when there was a default case.

Macintosh-specific Bugs

• Conversion operator member functions could not be implemented as trap functions; now they can.

• Destructors for PascalObject classes referenced undefined variables of the form "__ptbl...".

Bugs fixed in the B1 release:

• When local classes were used, certain virtual tables and functions were not emitted in the intermediate C code. This resulted in undefined symbols in the link. (A local class is a class, struct or union where the class type itself is declared within a function. Declaration of an object of class type is NOT a local class.)

• Names beginning with two underscores and an uppercase letter (__[A-Z]) are reserved (AT&T Product Reference Manual §2.4). Use of such names can cause CFront to generate incorrect code and/or crash. In this release, these names are now flagged with a warning. Similarly, class names should not contain a double underscore (no warning is produced in this case).

There may be other cases where a double underscore causes a problem in the CFront or C phases of the compilation. In this release, names containing a double underscore should be avoided if at all possible. (Note that §2.4 of the Language Reference Manual states that "identifiers containing a double underscore are reserved for use by C++ implementations and standard libraries, and should be avoided by users.")

• The *MPW 3.1 B* versions of **CLib881.o** and **CType.h** are now included with *MPW C++ 3.1 B1*, as the spelling of one of the internal data structures was changed, and could cause unresolved reference errors at link time, for *MPW 3.0* users.

Known Bugs and Inconsistencies

• There are still a few areas where CFront is either badly broken or seriously wounded. These areas include:

Inline Functions -- there are many limitations concerning what is

permissible within an inline function.

Inefficient Code Generation -- most cases of inefficient code result from the fact that semantic information is lost in the translation to C. This should be viewed as evidence that C is a poor choice of intermediate languages, rather than that C++ inherently produces poor code.

In addition to these general areas, there are still many minor inconsistencies between C++ as described in the AT&T Language reference manual and C++ as implemented by CFront 2.0. Remember, C++ is a young language and CFront is an inherently imperfect implementation of the language.

• Include files are searched for in the current directory first, not in the directory containing the source file.

• *MPW C++* does not allow "incomplete types" to appear in classes, structs or unions (neither does ANSI C). This may cause some difficulty for MPW C users who are able to write:

```
struct T {
        int a;
        int intarray[];         /* incomplete type! */
};
```

to describe data types that end with a variable length array.

• *MPW C++* currently "narrows" floating point return values in functions whereas *MPW C* leaves the return value as an extended value. This inconsistency may cause some difficulty when mixing functions written in C and C++ that return non-extended floating point values. We anticipate that future releases of *MPW C++* will follow the convention used by *MPW C.*

• **PascalObject** Limitations and Peculiarities

1. When using object hierarchies based on the built-in class **PascalObject**, the file containing the main entry point, i.e. "main()", must contain at least one class derived from **PascalObject**, in order for the **PascalObject** initialization code to be called. Something like the following will suffice in cases where no other references would appear.

```
class foo : PascalObject {};
```

Note that this precludes writing your main entry point in C. Pascal or Assembler may be used if a reference to an Object Pascal object is similarly referenced.

2. Pure virtual functions are not allowed in **PascalObject** hierarchies. Currently, CFront will accept the pure virtual declaration syntax, but the linker will fail with an unimplemented function error.

3. Pointer-to-members in **PascalObject** hierarchies are not supported due to implementation differences between C++ and Object Pascal. (CFront currently does not complain about creating or using such a pointer-to-member, but the resulting application will either fail to link properly or will fail at run-time with an error in the method dispatcher.)

4. The **pascal** keyword is broken in the specific situation where one attempts to call a C function which returns a pointer to a Pascal-style function. The C compiler currently misinterprets the C function as a Pascal-style function and the function result is lost.

5. The *MPW C++* keyword **inherited** can only be used within **PascalObject** hierarchies. This is a design decision, not a bug (We are trying to avoid gratuitous changes to C++, except where necessary for compatibility with the Macintosh, here specifically MacApp.)

• Incorrect code generation:

1. Using pointers to member functions of a class as default parameters to member functions of the same class sometimes causes problems. Using a

member function as a default argument to a subsequent member works:

```
struct A {
      void foo();
      void bar(void (A::*pfoo)() = &A::foo);
};
```

However, using a member function as a default argument to a previous member does not work:

```
struct A {
      void bar(void (A::*pfoo)() = &A::foo);
      void foo();
};
```

A workaround does exist, however, for this case:

```
struct A;

extern void (A::*pfoo)();

struct A {
      void bar(void (A::*pf)() = pfoo);
      void foo();
};

void (A::*pfoo)() = &A::foo;
```

You could also use the Apple extension that a variable can be declared "extern" and then "static", so the last line would become:

```
static void (A::*pfoo)() = &A::foo;
```

2. Statements of the form:

```
A a[2] = {A(opt_parms), A(opt_parms)};
```

where A is a class, fail, since the constructors for the objects being assigned into the array are never called.

3. Statements like:

```
int a;
int b;
a = b = 0;
```

still produce the erroneous "b used before set" warning.

4. There is a problem with some forms of inline operators. If an inline is declared to return a value and a path through the function does not, CFront may expand the function to just return "0". It may not warn you about this.

For example:

```
inline int operator --(stack& s) {
        if (size > 0) return *top;
        else
                cout << "Error";
}
```

The path producing an error does not explicitly return and CFront may simply generate an expression that evaluated to 0 in that case.

5. In the following example, CFront can coerce a B* to an A* for f(), but cannot coerce a B** to an A** for ff().

```
struct A { };
struct B : A { };
```

```
void f(A*);
void ff(A**);

B *bp, **bh;

void g()
{
        f(bp);
        ff(bh);
}
```

6. In the following example, CFront incorrectly reports that it cannot access operator new() from the private base class Base.

```
class Base : HandleObject {};
class Derived : private Base { };

main()
{
        Derived *dp = new Derived;
}
```

This problem seems to be limited to classes derived from HandleObject. As a workaround, override operator new in the Derived class to call HandleObject::operator new.

Macintosh® Programmer's Workshop C++ Reference

Printed in the United States of America.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014-6299
(408) 996-1010

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, APDA will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to APDA.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures and tables

Preface

MPW® C++ is an implementation of the C++ programming language, with extensions for programming in the MPW 3.1 environment on the Apple® Macintosh® computer.

This reference provides information about the Macintosh Programmer's Workshop (MPW) C++ Translator. The C++ programming language, developed by Bjarne Stroustrup, is a superset of the C programming language developed by Brian W. Kernighan and Dennis M. Ritchie.

Intended audience

This manual is intended for programmers who want to use MPW C++ to write Macintosh applications, build Macintosh tools (programs that run under the MPW environment), write applications that use both MPW C++ and Object Pascal in the source code, or write applications using the MacApp® libraries.

This manual assumes you are an experienced C programmer and are familiar with the following:

- MPW 3.1
- MPW 3.1 C
- C++ programming language

What you need

To use MPW C++ you need the following software products and accompanying documentation:

- MPW 3.1
- MPW 3.1 C
- MPW C++

If you intend to use MPW C++ with Object Pascal, you need the MPW 3.1 Pascal product.

If you intend to use MPW C++ with MacApp, you need the MacApp 2.1 product and MPW 3.1 Pascal.

The minimum system configuration for using MPW C++ is a Macintosh Portable, Macintosh Plus, Macintosh SE–family or Macintosh II–family computer, with a hard disk and 2 megabytes (MB) or more of memory, running MPW 3.1. You might need to disable MultiFinder® if you do not have enough memory.

A reasonable configuration for developing applications with MPW C++ is a Macintosh II–family computer with a hard disk, 4 MB or more of memory, and System 6.0.4 or later software, with MultiFinder. If you use MPW C++ with MacApp, you might need 5 MB or more of memory.

What is in this manual

This manual includes the following:

- Chapter 1 contains an overview of the C++ programming language and MPW C++.
- Chapter 2 contains instructions for installing MPW C++, and for building the sample programs that come with MPW C++.
- Chapter 3 describes the MPW 3.1 extensions to C++ for the Macintosh environment.
- Appendix A describes the compiler options you can use with MPW C++.
- Appendix B contains a list of the MPW C++ keywords.
- Appendix C documents the Complex Mathematics library distributed with MPW C++.
- Appendix D documents the Stream library distributed with MPW C++.
- Appendix E suggests ways to make your MPW C++ more robust, readable, and portable.

The MPW C++ 3.1 package also includes the following publications from AT&T:

- *UNIX System V AT&T C++ Language System Release 2.0: Product Reference Manual.* This publication is the C++ language reference manual. It supersedes the "Reference Manual" appendix in Stroustrup's *The C++ Programming Language.* Do not use the first edition of Stroustrup's book as a reference. (The second edition, which will describe release 2.0 of AT&T C++, is in progress.)
- *UNIX System V AT&T C++ Translator Release 2.0: Library Manual.* This publication describes the C++ libraries, including the Complex Mathematics and Stream libraries. Appendixes C and D of the present manual describe special features of the MPW C++ versions of these libraries.
- *UNIX System V AT&T C++ Language System Release 2.0: Selected Readings.* This publication contains articles written by Bjarne Stroustrup and his colleagues. These articles describe the language features that were added to the C++ programming language after 1985. (These features are not documented in Stroustrup's book *The C++ Programming Language,* first published in 1986 and reprinted with corrections in 1987. They are, however, documented in Lippman's *C++ Primer,* cited later in this Preface.)
- *UNIX System V AT&T C++ Language System Release 2.0: Release Notes.* This publication describes features new to version 2.0 of AT&T C++, as well as the (irrelevant) installation procedure for C++ on a UNIX® system.

How to use this manual

This manual assumes that you are already familiar with the C++ language. If not, you should first read an introductory such as Lippman's *C++ Primer*.

Chapter 1, "Introduction to MPW C++," gives an overview of the C++ programming language.

Read Chapter 2, "Using MPW C++," for instructions on how to install and run MPW C++ on your Macintosh computer.

Read Chapter 3, "MPW C++ Language Extensions," for information regarding the Apple Computer extensions to C++ that enable you to use C++ in the Macintosh programming environment.

Use the *UNIX System V AT&T C++ Language System Release 2.0: Selected Readings* to refer to the articles written by Bjarne Stroustrup and his colleagues about the evolution and features of C++. These articles contain C++ concepts and programming examples. They are part of the AT&T manual set for the latest version of the CFront translator, and you can use them to help you understand how to use the C++ programming language. The C++ language itself is defined in the *UNIX System V AT&T C++ Language System Release 2.0: Product Reference Manual*. If you are already familiar with version 1.2 of C++, you should read Stroustrup's paper "The Evolution of C++," in the *Selected Readings*.

Required documentation

You need these reference materials:

- *Macintosh Programmer's Workshop 3.0 Reference* and *MPW 3.1 Release Notes*. APDA™, 1988–89. These describe the Macintosh Programmer's Workshop 3.1 environment in which the MPW C++ Translator operates, including the editor, linker, debugger, and other important tools.
- *Macintosh Programmer's Workshop 3.0 C Reference* and *MPW C 3.1 Release Notes*. APDA, 1988–89. These describe the Macintosh Programmer's Workshop 3.1 C product, and the Pascal and C calling conventions.

You also need this book:

- *C++ Primer*. Stanley B. Lippman. Addison-Wesley, 1989. An introductory textbook on C++ programming. This is the most complete and comprehensive book now available on C++, including features recently added to the language.

Recommended documentation

The following documentation may be useful to you, depending on the type of application you plan to develop.

- *The C Programming Language*. Second edition. Brian W. Kernighan and Dennis M. Ritchie. Prentice-Hall, 1988. The standard reference book for the C language, rewritten for draft proposed ANSI C.
- *The C++ Programming Language*. First edition. Bjarne Stroustrup. Addison-Wesley, 1986. This is the classic book on C++, by its designer, which describes the C++ language as it was in 1985. It was the standard reference for the C++ programming language before release 2.0 of AT&T C++. It is not up to date, however, because Stroustrup has added a number of important language features since this book was published. The first edition of this book is largely of historical interest, although it still has the best explanations of certain concepts in C++ and has some useful examples. *Do not use the first edition as a reference.* (The second edition, which will describe release 2.0 of AT&T C++, is in progress.)
- *Inside Macintosh*. Volumes I–III. Addison-Wesley, 1985. Contains information you need in order to program by using the Macintosh ROM and associated RAM routines; these volumes cover windows, alert boxes, menus, graphics, and much more. Volumes I through III apply to all Macintosh computers.
- *Inside Macintosh*. Volume IV. Addison-Wesley, 1986. Describes the 128K ROM routines available with the Macintosh Plus or Macintosh 512K enhanced. (Some of these routines, such as the hierarchical file system routines, are also available on disk for machines that have the 64 KB ROM.)
- *Inside Macintosh*, Volume V. APDA, 1987. Describes the ROM routines available with the Macintosh II and Macintosh SE. (Some of these routines, such as the Styled TextEdit functions, are also available on the Macintosh Plus.)
- *MacsBug Reference*. APDA, 1988. Describes MacsBug, the object-level debugger from Apple Computer, Inc.
- *SADE Reference*. APDA, 1988. Describes SADE®, the source-level debugger from Apple Computer, Inc.
- *ResEdit Reference*. APDA, 1988. Describes ResEdit™, the resource editor from Apple Computer, Inc.
- *Apple Numerics Manual*. Second edition. Addison-Wesley, 1986. For the programmer who wants more understanding or control of the underlying floating-point arithmetic in MPW C. This manual describes the Standard Apple Numeric Environment (SANE®), which includes extended-precision floating-point arithmetic as specified by IEEE Standard 754. It describes each routine in detail, including boundary conditions and exception handling, and explains how to control the floating-point environment.

- *Motorola MC68000 32-Bit Microprocessor User's Manual.* Second edition. Prentice-Hall, 1985. Describes the Motorola MC68000 processor in detail for hardware and software engineers.
- *Motorola MC68020 32-Bit Microprocessor User's Manual.* Second edition. Prentice-Hall, 1987. Describes the Motorola MC68020 processor in detail for hardware and software engineers.
- *Motorola MC68030 32-Bit Microprocessor User's Manual.* First edition. Prentice-Hall, 1988. Describes the Motorola MC68030 processor in detail for hardware and software engineers.
- *MC68881/882 Floating-Point Coprocessor User's Manual.* Motorola, Inc., 1988. Describes the instruction set and addressing conventions used by the Motorola MC68881 and MC68882 floating-point coprocessors, which are used in the Macintosh II family of computers.

Several books have come out recently. They describe the C++ language of today, and show how to use language features that Stroustrup's 1986 book does not. This list is not exhaustive, because new books are coming out rapidly. These books do not describe the very latest additions to the C++ language, because the language changed after the books went to press. (This comment will apply to all C++ books, because the language continues to evolve.)

- *C++ for C Programmers.* Ira Pohl. Benjamin/Cummings, 1989. An introductory textbook on C++ programming for those who are already fluent in C. It complements Ira Pohl and Al Kelley's *A Book on C* (Menlo Park, California: Benjamin/Cummings, 1984).
- *Programming in C++.* Stephen C. Dewhurst and Kathy T. Stark. Prentice-Hall, 1989. An introductory textbook on C++ programming.

Conventions

This manual uses the following conventions.

- Source code is in a `monospace` font:

```
struct complex{
   extended re;
   extended im;
};
```

- Replaceable items (nonterminal identifiers) in syntax descriptions are in *italic:*

  `CPlus` *`mysource`*`.cp`

  Italics are used in this case to denote an MPW convention that is commonly used, but optional.

Chapter 1 **Introduction to MPW C++**

This chapter introduces the MPW® C++ implementation of the C++ programming language developed by Bjarne Stroustrup. Refer to the *UNIX System V AT&T C++ Language System Release 2.0: Product Reference Manual* for the official definition of the C++ language and its operation at the time of publication.

The first part of this chapter describes the language as an extension to the C programming language. The second part of this chapter briefly describes the features that were added to C++ after 1985. These features are discussed in detail in the AT&T publications listed in the Preface.

What is C++?

C++ is a programming language with features and facilities in common with other programming languages like Fortran, Pascal, and C. Like those languages, it is a high-level programming language. Unlike those languages, C++ is designed to support data abstraction and (OOP).

At AT&T Bell Laboratories, Bjarne Stroustrup worked with the creators of the C programming language, Brian W. Kernighan and Dennis M. Ritchie, to extend the power and flexibility of C while keeping its best features. C++ is the result. One can view C++ as a set of improvements to C, many of them similar to ANSI C, as well as features to support data abstraction and object-oriented programming.

C++ is an evolving language. MPW C++ is based on AT&T CFront 2.0, the version of the C++ translator that supports additional features like multiple inheritance and type-safe linking.

Improvements to C

Stroustrup chose C as the foundation of C++ because it is "versatile, terse, and relatively low-level" and because it runs in a number of programming environments and has been used (and shown to work well) for large, complex system-programming projects.

C++ differs from ANSI C, which also improves on Kernighan and Ritchie's original C, because the charter for ANSI C was to standardize extensions already in various dialects of C; C++, on the other hand, brings in additional features from other languages.

Perhaps the most important extension to C is the creation of a new data type, the *class*. With classes it is possible to program using data abstraction, data hiding, and inheritance—features required for object-oriented programming.

The C++ programming language is a superset of C and uses the same libraries as C. Stroustrup added more than new features to the language, however; he also added new concepts to C programming. C++ supports object-oriented programming with type derivation and data abstraction.

In addition to these conceptual differences from C, C++ also has many other new features and enhancements. For more information, refer to the *UNIX System V AT&T C++ Language System Release 2.0: Product Reference Manual.* This section briefly summarizes these new additions to the C programming language:

- the `//` comment delimiter
- name space changes
- function prototypes and argument type checking. (This feature is also in ANSI C.)
- default parameter values
- overloaded functions
- inline functions
- reference variables

The // comment delimiter

In C++ you can comment to the end of the line using the double forward slash, `//`.

```
// This is a C++ comment
```

Name space changes

There are fewer name spaces; tags for `enum` constants and `struct` constants are automatically type names so that the following is no longer necessary:

```
typedef struct foo foo;  //make foo a typedef for the struct foo
```

Function prototypes and argument type checking

Like ANSI C, C++ has function prototypes and strong type checking. For example, in the header file complex.h,

```
struct complex  {
   extended re;
   extended im;
};
complex cadd  (complex x, complex y)
```

the function `cadd`, as defined, accepts two `complex` numbers as parameters and returns a `complex` result. The following code would normally give an error message because `cadd` is declared as a function with two `complex` arguments, and the example uses a `float` argument:

```
#include <complex.h>
complex a, b;
float c;
...
a = cadd(b,c);
```

However, C++ allows the user to specify type conversions, and the complex.h header has rules for converting `float` values into `complex` values. Conversion operators are introduced later in this section. For more information on function prototypes and argument type checking, see the *UNIX System V AT&T C++ Language System Release 2.0: Product Reference Manual.*

Default parameter values

C++ also allows specification of default values for parameters. For example,

```
float Wages (float PayRate, float Hours, float ShiftDifferential = 1.0)
```

specifies a function `Wages`, which can be called as follows:

```
Wages(12.50, 8.0);   //day shift,no shift differential = 1.0
Wages(12.50, 8.0, 1.2); //swing shift,20% shift differential
Wages(12.50, 8.0, 1.4); //graveyard shift,40% shift differential
```

For more information on default values for parameters, see the *UNIX System V AT&T C++ Language System Release 2.0: Product Reference Manual.*

Overloaded functions

Functions can be *overloaded.* This means there can be multiple versions of a function with the same name accepting different argument types. The appropriate version is called, depending on the actual arguments in each call.

For example, using the `complex` definitions in the previous section, there could be two versions of `cadd`:

```
complex cadd (complex x, complex y);
complex cadd (complex x, float y);
```

so that

```
#include <complex.h>
complex a, b;
float c;
...
a = cadd(a,b); //uses cadd(complex x, complex y)
a = cadd(b,c); //uses cadd(complex x, float y)
```

There can be any number of overloaded versions of a function, provided that the parameter types can be distinguished from one another. (The parameter types must differ between versions, whether or not the return types differ.) For more information on overloaded functions, see the *UNIX System V AT&T C++ Language System Release 2.0: Product Reference Manual.*

Inline functions

The `inline` mechanism allows the specification of small procedures as inline C++ code. For example, using the `complex` definitions in the previous section,

```
inline extended norm (complex a) {return a.re*a.re + a.im*a.im;}
```

specifies the `norm` function in such a way that the C++ compiler can emit inline C code. The `inline` keyword is purely advisory, like `register` in C. The compiler decides what can be expanded inline and what cannot.

Inline functions have advantages over macros provided by the `#define` of the C preprocessor, in that scope rules apply and argument types are checked. For more information on inline functions, see the *UNIX System V AT&T C++ Language System Release 2.0: Product Reference Manual.*

Reference variables

A *reference* variable is one that does not actually contain a value, but refers to another variable containing the value. Whenever the program mentions a reference variable, the variable to which it refers is used instead. (Note that this is different from pointers, which must be explicitly dereferenced to get the value pointed to.)

The most obvious use of references is to declare call-by-reference parameters to a function. When a function parameter is a reference, any use of the parameter refers to the actual argument. When the function "modifies" the reference parameter, the actual argument is changed.

For example:

```
void increment(int &x)
{
   x++;
}
...
int y = 1;
increment(y);  // y becomes 2
```

For more information, see the *UNIX System V AT&T C++ Language System Release 2.0: Product Reference Manual.*

Support for data abstraction

C++ has a number of new language features that support data abstraction, data hiding, and the building of interface files for compiled libraries. These include

- member functions (functions associated with `struct` types)
- classes
- constructors and destructors
- operator functions
- user-specified data conversions

Member functions of structures

In C++ you can declare functions as well as data items to be components of a `struct`. Here is an example:

```
struct stack {
   int top;
   int elements [MAX];
   void push (int);
   int pop();
};
```

In this example, in which a stack is implemented as an array, the data item `top` points to the current top of the stack and the array `elements` contains the actual stacked data; `push` and `pop` are the usual stack functions. A program using this structure declaration can create several stacks:

```
stack s1;
stack s2;
   ...
s1.push(5);
s2.push(17);
```

The intention is that `push` and `pop` are the interface to be used for stack manipulations, and it is generally assumed that only `push()` and `pop()` will use `s1.top` and `s1.elements`; programs that use stacks will access them with `push()` and `pop()`. Following these conventions would permit the structure shown above to be replaced, for example, by an entirely different one in which the stack is implemented as a linked list. As long as the interface functions `push` and `pop` are rewritten so that they operate correctly on the new data structure, the user of these functions need not be aware of the change in implementation.

Classes

The distinction between a `struct` that contains member functions and a `class` is one of data protection. The stack example of the previous section could have been written as

```
class stack {
private:
   int top;
   int elements [MAX];
public:
   void push (int);
   int pop();
};
```

The difference between the two examples is that in the latter case a client program is not permitted to write

```
stack s1;
s1.elements[3] = 2;
```

The `private` data are only accessible to the member functions `push` and `pop`. There is also an intermediate level of protection called `protected` available for class members. For further information on data protection, see Chapter 7, "Access Rules for C++," in *UNIX System V AT&T C++ Language System Release 2.0: Selected Readings.*

A class can be viewed as a programmer-defined type, in which the programmer has defined a data structure and the operations permissible on it.

Constructors and destructors

C++ provides constructors for creation and initialization of structures and classes, and destructors for their termination. A constructor is a member function with the same name as a class. It is automatically executed after storage is allocated and before the first use of the class variable. A destructor is a member function with the name of the class preceded by a tilde (~). It is automatically executed before storage is freed, or at the point where a storage variable leaves scope: for instance, upon leaving a function.

Operator functions

Most of the built-in operators such as * and + can be overloaded using *operator functions.* An operator function is written just like an ordinary function, except that the function name specifies an operator:

```
complex operator+ (complex x, complex y) {return cadd(x,y);}
```

Whenever the operands of the + operator are both `complex`, this function will be called. As with other functions, implicit conversions will be used as needed.

User-specified data conversions

When a constructor having only one argument has been defined within a class (or structure), it is regarded as defining a conversion between the type of the argument and the class itself, the latter being a user-defined type. The inverse transformation, from the class to another type, is defined by including in the class a special kind of operator called a *conversion operator.* For more information on user-specified conversions, see the *UNIX System V AT&T C++ Language System Release 2.0: Product Reference Manual.*

Support for object-oriented programming.

The distinction between *data abstraction* and *object-oriented programming* is that data abstraction gives the programmer the ability to define and use new data types; object-oriented programming adds the ability to create a *derived* type from an existing user-defined type. The derived type implicitly has as members all of the members of its *base* (the type from which it was derived), and it may define additional data and function members. In this sense a derived type extends the base type. It may also override function members of the base type by redefining them. In this sense, it modifies the base type. The binding takes effect by default at compile time, because the choice of function is determined by the type of the variable or pointer.

There also exists a mechanism for run-time binding of overridden functions. The keyword `virtual`, applied to a function in a base class (or structure), tells the C++ compiler that the member function for the actual object type (as it was created) is to be used. In the case of virtual functions, selection can be deferred until run time if the class variable is accessed through a pointer, because the actual object type may differ from the pointer type.

A pointer variable that is declared to be a pointer to a base class can have the address of a derived class object assigned to it. If the derived class has an overriding function declaration that is declared as `virtual` in the base class, then selection of that function via the pointer causes a call to its derived class version. This selection takes place at run time.

Because a base class may have any number of derived classes, and any of these may be a base for a further derivation, a group of such classes forms a hierarchy. This is termed *single inheritance.* A derived type *inherits* the members of its base.

Multiple inheritance allows derivation of a class from more than one base class. In general, multiple inheritance allows the combination of independent concepts represented as classes into a composite concept represented as a derived class. A multiple-inheritance class structure is usually referred to as a hierarchy also, even though it is technically a Directed Acyclic Graph (DAG).

Language features new to Release 2.0

The C++ language in its current form includes a number of features that did not exist when *The C++ Programming Language* was written. For details refer to the articles in *UNIX System V AT&T C++ Language System Release 2.0: Selected Readings,* in particular Chapter 1, "Evolution of C++," as well as the *UNIX System V AT&T C++ Language System Release 2.0: Product Reference Manual.* Here are the features, in brief.

- Multiple inheritance—the ability to derive a class from more than one base class.
- `protected` class members—accessible to member functions of derived classes, but not accessible externally.
- Order of base and member initialization—formerly defined by the implementation. Now there is a definitive rule for the order in which initializations occur.
- Memberwise assignment and initialization of class objects—formerly done by bitwise copy. Now there is a correct, recursive, member-by-member assignment or initialization. (This is optimized to a bitwise copy where appropriate and safe.)
- Overloading of the `new`, `delete`, and `->` (arrow) operators—specific provision has been made for overloading of these operators. The first two enable programmer replacement of the default storage management methods for a given class. The last enables the defining of *smart* pointers.
- Pointers to members—a notation has been provided for describing a pointer to a member (a function or a data member) of a given class.
- Overloading sensitivity—the set of basic types that can be used as discriminants between members of a group of overloaded functions, increased from earlier versions of C++. For example, the overloading mechanism can now distinguish between signed and unsigned values, and between single-precision and double-precision floating-point values.
- Abstract classes—classes that can only serve as base classes for other classes. An abstract class cannot be instantiated, but its subclasses can.
- Static members of a class—a static data member is a member for which there is one copy for the class rather than one copy per object; a static member function is not a member of any object in the class, so it needs to be called using the special member-function syntax. Similarly, it does not have a `this` pointer, so it can only access non-static members of its class by using `.` or `->`.
- Local classes, `struct` constants, unions, and `enum` constants—classes, `struct` constants, unions, and `enum` constants whose class types are declared inside a function.

Why use C++?

C++ is a superset of the C programming language. It offers compatibility with C and the efficiency of a standard compiled and linked run-time environment. You can link existing C libraries with your C++ programs.

The object-oriented features of C++ let programmers work with data types designed specifically for the programming task at hand. On a large project, programmers can share libraries of created and derived classes rather than reinventing structures.

At the same time, C++ retains the features of C that account for its wide use in professional programming projects: support for high-level program structuring and low-level access to data and hardware. C++ also preserves the C-style modular interfaces to functions, global data, and external header files.

C++ is suitable for a wide range of applications. The techniques of data abstraction and object-oriented programming lend themselves particularly well to the interactive graphic user interface on the Apple® Macintosh® computer.

About MPW C++

MPW C++ is implemented using CFront 2.0, a translator (licensed from AT&T) that accepts C++ source code as input and produces C source code as output. MPW C++ uses the MPW 3.1 Linker and the various MPW 3.1 libraries, augmented by libraries that are specific to C++. MPW C++ requires the MPW 3.1 environment.

MPW C++ supports the Stream libraries that are distributed by AT&T as part of its C++ product. The numeric type `complex` is supported by an interface and library that serves the needs of both MPW 3.1 C and MPW C++.

The components of MPW C++ are

- CPlus, an MPW Shell script that invokes CFront, then invokes the MPW C compiler to produce linkable output
- CFront, a modification of the AT&T C++-to-C translator
- the MPW C++ libraries: CPlusLib.o, CPlusLib881.o, CPlusOldStreams.o, and CPlusOldStreams881.o

- the Unmangle tool, which can be used to translate a function name with a type signature into a human-readable string
- the MacsBug Demangler resource (which can be pasted into MacsBug with ResEdit™), which will demangle function names in MacsBug output

MPW C++ is distributed with three example programs you can build and run:

- an application (CPlusTESample), written in MPW C++
- an application (CPlusShapesApp), written in MPW C++
- an MPW tool (Count), written in MPW C++

The procedures for building an executable program are documented in Chapter 2, "Using MPW C++."

Apple Computer, Inc. has added the following language extensions to this implementation of MPW C++ to provide programming support in the MPW environment for the Macintosh computer:

- support for accessing the Macintosh operating system and toolbox
- support for code development using Object Pascal and MacApp®
- support for the Standard Apple Numeric Environment, SANE®
- support for casting pointers to member functions in multiple-inheritance hierarchy (Although part of the language, this feature is not supported by AT&T CFront 2.0.)

The MPW C++ language extensions are documented in Chapter 3, "MPW C++ Language Extensions."

Chapter 2 **Using MPW C++**

This chapter describes how to install MPW C++ on your Macintosh, and how to build and execute sample C++ programs. The sample programs include source code for an MPW tool, and two C++ object-oriented programs. This chapter also describes how to compile your programs from the command line, and it documents the libraries to use in your link list, which vary depending on the type of program.

Getting started

This section explains how to install MPW C++, and how to build applications and tools.

Installing MPW C++

The following instructions assume you have installed MPW 3.1 and MPW 3.1 C on your hard disk. These installation procedures are documented in the *Macintosh Programmer's Workshop 3.1 Reference.*

To install MPW C++ on your hard disk, follow these steps:

1. Insert the *MPW Installation Disk* into your 3.5 inch drive.
2. Copy the folder named {Installation Folder} into your {MPW} folder on your hard disk. (Your MPW folder must be named "MPW.") Eject the *MPW Installation Disk.*
3. Open the folder {Installation Folder} and start the "MPW Installer" application.
4. Follow the screen prompt by inserting the *MPW CPlus Disk* into your 3.5 inch drive.
5. Click OK, and MPW C++ will be installed correctly into your MPW 3.1 folder.
6. Drag the folder {Installation Folder} to the Trash and empty the Trash. (Save it on the *MPW Installation Disk.*)

Alternatively, you can follow these steps. Be sure to open the folders and copy the contents, *not* the folders themselves: otherwise you will overwrite your existing MPW 3.1 folders.

1. Copy the CPlus.Help file to the {MPW} folder.
2. Copy the *contents* of the {Tools} folder to the {MPW}Tools folder.
3. Copy the *contents* of the {Scripts} folder to the {MPW}Scripts folder.
4. Copy the *contents* of the :Libraries:CLibraries folder to the {MPW}Libraries:CLibraries folder.
5. Copy the CPlusExamples folder to the {MPW}Examples folder.
6. Copy the *contents* of the :Interfaces:CIncludes folder to the {MPW}Interfaces:CIncludes folder.

This completes the installation of MPW C++ onto your hard disk. You are now ready to begin using the examples to compile and link the sample programs.

◆ *Note:* To use MPW C++, you must have MPW 3.1 and MPW 3.1 C. To use C++ with Object Pascal, you must have the MPW 3.1 Pascal product.

About the C++ examples

MPW C++ is distributed with three sample programs you can build and run:

- CPlusShapesApp, a minimal application written in MPW C++
- CPlusTESample, an application written in MPW C++
- Count, an MPW Tool written in MPW C++

The CPlusExamples folder contains an Instructions file and the appropriate files for the three sample programs. The Instructions file in the CPlusExamples folder contains the most up-to-date information about building the sample applications.

CPlusShapesApp is a minimal application that demonstrates how to initialize commonly used toolbox managers, operate under MultiFinder®, handle desk accessories, and open and close windows. CPlusShapesApp shows how a simple class hierarchy can be built to draw various shapes in a window. It uses a simple set of classes that demonstrate some of the advantages of object-oriented programming.

CPlusTESample is an example application that demonstrates how to initialize the commonly used toolbox managers, operate successfully under MultiFinder, handle desk accessories, create, grow, and zoom windows, and create and maintain scrollbar controls. CPlusTESample shows how an object-oriented program can be written. It uses a simple set of classes that demonstrate some of the advantages of object-oriented programming.

Count is a sample tool written in MPW C++. It counts characters and lines in a file and reports the information.

Using the Commando interface

The Commando user interface helps you create a CPlus command line using special Macintosh dialogs, rather than the traditional command-line interface. The Commando interface consists of several dialog boxes containing a variety of controls. The CPlus Commando dialog helps you

- select the source (`.cp`) files you want to compile
- select command-line options
- select options controlling the output of virtual tables and method tables
- set debugger options
- set directories to search for `#include` files

- name a file or device other than the MPW worksheet to receive output from the CPlus command
- select options to predefine `#define` symbols or undefine the predefined ones

To get information about an option, press and hold the mouse button while the pointer is on the option.

For complete information about the Commando interface, refer to your *MPW 3.0 Reference.*

To invoke the Commando interface for CPlus, type one of the following commands on the MPW worksheet:

```
Cplus<Option-Enter>
Cplus<Option-semicolon><Enter>
Commando Cplus<Enter>
```

(The key combination Option-semicolon displays an ellipsis [...] after the command name on your screen.)

Getting help

A brief description of each CPlus compiler option is available within the MPW 3.0 Shell.

To invoke help for CPlus, type

```
Help -f {MPW}Cplus.help CPlus<Enter>
```

The MPW 3.1 Shell displays a list of the MPW C++ command-line options, with a brief description of each.

Programming in MPW C++

MPW C++ is suitable for stand-alone applications and MPW tools. It is not, however, suitable for drivers, desk accessories, or other stand-alone code resources, because MPW C++ virtual functions allocate global storage, whose integrity cannot be guaranteed.

Compiling an MPW C++ program

To compile your program, first start the MPW 3.1 Shell application. Then enter the CPlus script command in the following form:

`CPlus` *mysource*`.cp<enter>`

where *mysource*.cp is the filename of your C++ source program.

The extension `.cp` is a Macintosh convention to distinguish MPW C++ source from MPW 3.1 C source: this convention is commonly used, but is not enforced. If you are using the CreateMake function to generate a makefile, use `.cp` as the extension to your CPlus source file. If you are not using CreateMake, you can use `.c` as the extension, but it is not recommended. This is because in the Make tool, the default rules for building object files assume that the `.c` suffix indicates that the C compiler should be used.

MPW is not case sensitive, so your filenames can be uppercase, lowercase, or mixed case.

The CPlus script creates the object file *mysource*`.cp.o`.

If you enter the command `CPlus` the compiler reads from standard input. This means that the compiler reads any text that you subsequently enter. This feature allows you to run the compiler interactively. You can tell that the compiler (rather than the MPW 3.1 Shell) is reading the text you enter because the name CFront appears in the status box in the lower-left corner of the active window.

Once the compiler is running interactively, you can enter source code in any window, composing your program as you go. Press the Enter key after each line of C++ code to send the code to the compiler. To terminate input, press the Command and Enter keys simultaneously. When the compiler compiles standard input, it creates an object file named `cp.o`.

A complete specification of the CPlus command-line options appears in Appendix A, "MPW C++ Compiler Options."

Building an application or tool

Compilation is an important part of building a program, but it is not the whole story. When you build an application or tool with MPW, several events take place:

1. You or MPW decides which object file depends on which source file, and a list of dependencies is created. This list forms the basis for the makefile, which controls steps 2 through 5.
2. Each source file is compiled or assembled.
3. Each object file is linked.
4. Resource description files are processed by Rez.
5. An executable file (application) or MPW tool file is created.

Often, the easiest way to build any program is to use the Build menu in the MPW 3.1 Shell. Because the makefiles already exist for the sample applications, you do not need to create them. You can use the following steps to build executable programs for the three examples: Count, CPlusTESample, and CPlusShapesApp. For complete information on creating makefiles and building applications, see your *MPW 3.1 Reference.*

To build one of the sample applications, follow these steps:

1. Start the MPW 3.1 Shell application.
2. Using the Directory menu, change to the directory that contains the source code for the application you want to build.
3. Choose the Build command from the Build menu.
4. Type the name of the executable file you want to build. Use the name of the makefile, but do not type the suffix `.make`. The system will display information showing you the time, the commands that are executing, the libraries being linked, and the name of the resulting executable file.
5. Press Enter to launch the sample application.
6. Choose the Quit command from the File menu to return to the MPW 3.1 Shell.

MPW C++ interface libraries

MPW C++ programs can use the libraries supplied with MPW 3.1 C. These libraries are documented in the *MPW 3.0 Reference* and the *MPW 3.0 C Reference.* The calling conventions to use with these libraries are documented in Chapter 3 of this book, "MPW C++ Language Extensions," and in Appendix C, "Calling Conventions and Type Correspondence," of the *MPW 3.0 C Reference.*

MPW C++ supports the numeric type `complex` and the Complex Mathematics and Stream libraries that are distributed by AT&T as part of the CFront 2.0 product. These libraries are documented in Appendix C, "MPW C++ Complex Mathematics Library," and Appendix D, "MPW C++ Stream Library," and in the *UNIX System V AT&T C++ Translator Release 2.0: Library Manual.*

The conventions in the *MPW 3.0 C Reference* are extended as documented in "Linking an MPW C++ Program with Libraries," later in this chapter. To link MPW C++ programs you need all the libraries provided with MPW 3.1 C, as well as the CPlus libraries. For example, {CLibraries}CPlusLib.o relies on routines defined in {CLibraries}StdCLib.o.

MPW C++ header file conventions

All the header files for MPW 3.1 C can be used with MPW C++. In the few places where the two languages differ, the symbol `__cplusplus` distinguishes them.

Some header files are specific to MPW C++. Refer to Appendixes C and D of this reference for a description of these files.

- *Note:* See "About the C Libraries" in Chapter 1 of the *MPW 3.0 C Reference* for a description of the libraries you access with the MPW 3.1 C and MPW C++ compilers. The C++ Complex Mathematics and Streams libraries are documented in Appendixes C and D of this reference.

Linking an MPW C++ program with libraries

The following tables show the libraries to use when you link an application or a tool written with MPW C++ in your source code. You use the same libraries for tools and applications, except that a tool might also need the library ToolLibs.o if it calls any of the spinning-cursor or error-manager routines. It is included in the library list for completeness.

Table 2-1 shows the libraries to use for code that is not using the MC68881 or MC68882 math coprocessors. Link a tool or application written in C++ with the libraries listed in Table 2-1.

■ **Table 2-1** Libraries for linking with C++

Macintosh interfaces	Run-time support	Standard C library	MPW C++ library
{Libraries}Interface.o	{CLibraries}CRuntime.o	{CLibraries}StdClib.o	{CLibraries}CPlusLib.o
{Libraries}ToolLibs.o	{Libraries}Stubs.o	{CLibraries}CInterface.o	{CLibraries}CPlusOldStreams.o
		{CLibraries}CSANELib.o	
		{CLibraries}Math.o	
		{CLibraries}Complex.o	

If, for compatibility reasons, you are required to use the old (obsolete) stream library, (CPlusOldStreams.o), you must place CPlusOldStreams.o in your link list before CPlusLib.o. This is because the old stream library must override the new stream library, which is contained in CPlusLib.o.

Table 2-2 shows the libraries to use when you compile with the `-mc68881` option. For code compiled to take full advantage of the MC68881 or MC68882 coprocessors on Macintosh II–family machines, link with the libraries in Table 2-2.

■ **Table 2-2** Libraries for linking with C++, using the `-mc68881` option

Macintosh interfaces	Run-time support	Standard C library	MPW C++ library
{Libraries}Interface.o	{CLibraries}CRuntime.o	{CLibraries}CLib881.o	{CLibraries}CPlusLib881.o
{Libraries}ToolLibs.o	{Libraries}Stubs.o	{CLibraries}CInterface.o	{CLibraries}CPlusOldStreams881.o
		{CLibraries}StdCLib.o	
		{CLibraries}CSANELib881.o	
		{CLibraries}Math881.o	
		{CLibraries}Complex881.o	

To use the 881 libraries, you must compile the C++ source with the `-mc68881` compiler option. Then you must link the 881 libraries *before* the non-881 libraries. The linker always takes the first definition encountered, and ignores the duplicate definitions resulting from having both the 881 and non-881 versions of the libraries in the link list. (You can use the `-d` option to suppress warnings about duplicate definitions.)

Remember to include CLib881.o before the other libraries in {CLibraries} because it contains some definitions that override 80-bit versions in StdCLib.o and CRuntime.o. See the *MPW 3.0 C Reference* for more details.

If, for compatibility reasons, you are required to use the old (obsolete) stream library (CPlusOldStreams881.o), you must place CPlusOldStreams881.o in your link list before CPlusLib881.o. This is because the old stream library must override the new stream library, which is contained in CPlusLib881.o.

Linking with MacApp

You will not often use both C++ and Object Pascal source files in the same application, but if you write a C++ application using MacApp, you will need to link both C++ and Object Pascal together. To do so, use the libraries listed in Table 2-3. (Most of these are the same as in Table 2-1, but there are differences in the first two columns.)

You need the library ObjLib.o if you derive from the predefined class `PascalObject` anywhere in your source code.

The same rules apply for the old stream libraries as mentioned in the previous section.

■ **Table 2-3** Libraries for linking with MacApp

Macintosh interfaces	Run-time support	Standard C library	MPW C++ library
{Libraries}Interface.o	{CLibraries}CRuntime.o	{CLibraries}StdCLib.o	{CLibraries}CPlusLib.o
{Libraries}ObjLib.o		{CLibraries}CInterface.o	{CLibraries}CPlusOldStreams.o
{PLibraries}PASLib.o		{CLibraries}CSANELib.o	
{PLibraries}SANELib.o		{CLibraries}Math.o	
		{CLibraries}Complex.o	

To link C++ code with Object Pascal code, and to take full advantage of the MC68881 or MC68882 coprocessors on Macintosh II family machines, link with the libraries listed in Table 2-4. (Most of these are the same as in Table 2-2, but there are differences in the first two columns.)

■ **Table 2-4** Libraries for linking with MacApp, using the `-mc68881` option

Macintosh interfaces	Run-time support	Standard C library	MPW C++ library
{Libraries}Interface.o	{CLibraries}CRuntime.o	{CLibraries}CLib881.o	{CLibraries}CPlusLib881.o
{Libraries}ObjLib.o		{CLibraries}CInterface.o	{CLibraries}CPlusOldStreams881.o
{PLibraries}PASLib.o		{CLibraries}StdCLib.o	
{PLibraries}SANELib881.o		{CLibraries}CSANELib881.o	
		{CLibraries}Math881.o	
		{CLibraries}Complex881.o	

The dump/load option

When large header files are used, the dump/load facility can greatly reduce compile times. The portion of the compilation that reads the header files can be eliminated except for a few seconds to read a large disk file.

The `-dump` *filename* option saves the state of the C++ compilation (mostly the compiler symbol table) to the specified file. A matching `-load` *filename* restores the state. In effect, the compilation is suspended and resumed—the dump portion of the compile is only done once, while the load portion can be repeated for many different source files.

For example, you could say

```
cplus header_file.h -dump header_file.h.dump
cplus source_one.c  -load header_file.h.dump
cplus source_two.c  -load header_file.h.dump
```

If the dump file name ends with a colon (`:`), it is suffixed with the source file name and `.dump`, so the first command is equivalent to

```
cplus header_file.h -dump :
```

The `-load` compilation can include additional headers if needed, but only one load file can be read during a compile. This also means that the load occurs before any `-d` or `-u` options are executed, so macros can be added or removed on the command line. This also means you can't use `-d` or `-u` to change the meaning of the code that was dumped. You can use `-d` or `-u` on your load to control the code that is seen after the point at which the load is done; you can't use them to alter the macros defined when the dump was done.

In other words, the `-load` compilation looks like:

source code that was compiled with -dump
`#define ...` *from* `-d` *options*
`#undef ...` *from* `-u` *options*
source code in this compilation

If you have global definitions in the header file, you will get a nontrivial header_file.h.o file, which you may want to include in your link. In particular, this may happen because of nontrivial `const` declarations such as

```
const char string[] = "abc";
```

Static objects in a header file will be compiled into the header_file.h.o file, where they will be inaccessible from other compiles, even one that uses `-load`. Because of this, static objects cause a compilation warning.

If your headers are protected with the `#ifndef...#define...#endif` convention, you can leave your `#includes` in the source file, because if a preprocessor symbol was there for the dump it will be there for the load. But you may wish to put a `#ifndef...#endif` around the line containing the `#include` itself in the source file, so that you don't waste time reading the header files just to skip them.

To generate a compressed dump file, use `-dumpc` (`-load` is used for both compressed and uncompressed versions). This reduces the file size by about 40 percent, with very little change in compile speed on the Macintosh II or Macintosh SE. On machines where the CPU-to-disk speed ratio is higher than the Macintosh II, compressed loads may actually run faster than noncompressed.

The command line options `-mc68881`, `-elems881`, `-n`, and `-x` are not allowed when using `-load`. These options are restored to their values from the dump file.

Dump/load does not work with previous versions of the MPW C compiler.

The dump/load mechanism built into MPW C++ is designed to counteract a common problem in large-scale application development and in OOP, in general. The problem is that in order to compile one small source file (typically less than 20 KB) the compiler must first process a much larger body of header file information (22,000 lines or 725 KB for the MacApp and Toolbox headers). Since the information in the header files rarely changes, much of this overhead can be eliminated. Dump/load allows you to precompile the header files and then rapidly load them into the compiler at startup. Typical speed improvements are on the order of 2 to 3 times.

Step-by-step instructions

1. Decide which header files you want to precompile. Typically this will include your Application Framework (MacApp, for example), Toolbox Interface, and Language Library header files.
2. Create a new header file called MyAppDump.h, which will contain `#include` directives for all of header files you decided to pre-compile in step 1.

   Now, look through your application's source and header files, for `#include` references to any of the files from step 1. In each application file that contains a reference: paste a copy of the `#include` lines into MyAppDump.h and replace the `#include` lines in the source file with the following:

   ```
   #include "MyAppDump.h"
   ```

3. Bracket the body of MyAppDump.h with `#ifndef...#endif` so that it looks like this:

```
#ifndef __MyAppDump__
#define __MyAppDump__
        ...
#include <Quickdraw.h>   // to be precompiled...
        ...
#endif
```

4. Add a build rule to your makefile corresponding to the compile and dump of the precompiled headers:

```
MyAppDump.h.o ƒ  MyAppDump.h   etc.
    CPlus MyAppDump.h -dumpc MyAppDump.h.dump ∂
      -o MyAppDump.h.o {CPlusOptions}
```

5. Modify the build rules of your application's source files to include a load of the precompiled headers, and a dependency on the dumpfile itself:

```
MySrc.cp.o ƒ MySrc.cp MySrc.h MyAppDump.h.o
    CPlus MySrc.cp -load MyAppDump.h.dump ∂
      -o MySrc.cp.o {CPlusOptions}
```

   (actually the `.o` corresponding to the dump—Make doesn't yet handle dependencies for tools producing multiple output files.)

6. Finally, add the object file produced by the dump phase (here MyAppDump.h.o) to your link dependency and link list since it may contain definitions of global `const` objects, virtual tables, bodies of inline virtual functions, and such.

MultiFinder memory option

When the `-mf` option is used, MPW C++ will request MultiFinder memory when the MPW shell partition is not large enough to complete the compilation. Of course, the amount of MultiFinder memory available depends on what other applications are running, and C++'s use of this memory will reduce the amount of memory available for running other applications.

MPW C++ releases MultiFinder memory at the end of each compilation, even when interrupted with Command-period. However, after certain catastrophic errors (such as a bus error recovered via "`g stoptool`" in MacsBug), MultiFinder memory may not be released. If this occurs, you must quit the MPW shell to free the MultiFinder memory.

With the `-mf` option, you can keep the MPW shell partition small enough (say, 1 to 2 MB) to run other applications with MPW, and still be able to do large compilations without sizing up the MPW shell.

Marking your source files

The `-mark` option tells MPW C++ to install MPW shell marks into the source files. Only the file listed on the CPlus command line are marked; include files are unaffected. When you use `-mark`, no marks are put in your include files. This is because the include files are read so often that the compiler would slow down considerably if it had to mark them every time it read them.

The `-mark` option has four suboptions:

- `fcts` marks functions.
- `types` marks types (including classes.
- `data` marks global data items.
- `all` marks functions, types, and data.

The `-mark` option's syntax is similar to that of the `-sym` option.

The added C++ marks begin with "–" (option-underscore). Any existing marks beginning with "–" are removed, and the new marks are merged with the old non-C++ marks.

Since the marks are maintained in the resource fork of the source file, there is no danger of corrupting the text. The modification time of the file is not changed by the `-mark` option. (Actually, it's reset to its original value.) Marking does not work with source files containing `#line` directives.

Pragmas

Unrecognized pragmas (for now, all but `#pragma segment`) are now passed through to the C compiler. However, any pragmas encountered within a function or other file-level declaration are emitted before the declaration that contains them. Since there is not an exact one-to-one mapping of C++ constructs to C constructs, position-dependent pragmas may not produce the expected effect. In particular, C type declarations used to implement C++ types may be emitted out of order or not emitted at all, if not needed.

Demangling tools

Because a C++ function name can be overloaded, all function names consist of the name specified by the user and a suffix called a "type signature" that encodes the particular sequence of parameter types for each overloaded version. (In C++, every function is uniquely defined by its name plus the types of its parameters. In C, every function is just defined by its name.) The process of adding a type signature to a name is commonly called *mangling*.

The unmangle tool is an MPW tool that attempts to demangle the name passed to it on the command line. The command

`unmangle` *mangled_name*

yields the demangled name, or reports that the name was not a mangled one. This can be useful when deciphering error messages from the linker about unreferenced externals with mangled names.

- *Note:* This tool and the library described below are *not* identical to the AT&T demangler. For example the unmangle tool and library do not demangle local variable names.

The file MacsBug 6.1 Unmangler on the disk *MPW CPlus2* contains a resource that should be pasted into your MacsBug 6.1 Debugger Prefs file with ResEdit. This resource allows MacsBug 6.1 to demangle the function names produced by C++. Thus, instead of seeing mysterious and potentially unsettling names like

```
__ct__8TPN_ExprFPcN21
```

in the disassembled code, or in the "popup name selector window" (press Command-colon) you'll see

```
TPN_Expr::TPN_Expr(char*,char*,char*)
```

instead. CFront/C generates these MacsBug names by default or with the command line option `-mbg full`.

We have also included the unmangler in library form in the file unmangle.o. The function `unmangle()`, which decodes C++ mangled symbols (that is, a symbol with a type signature), behaves in the following manner:

```
int unmangle(char *dest, char *src, int count);
```

where:

`dest` = pointer to result buffer
`src` = pointer to mangled symbol
`count` = max chars to write not including null(e.g. sizeof(buffer) - 1)

The return codes are

–1 = error; probably because symbol was not mangled, but looked like it was
0 = symbol wasn't mangled; not copied either
1 = symbol was mangled; unmangled result fit in unchanged buffer
2 = symbol was mangled; unmangled result truncated to fit in buffer (null written)

Implementation details

Here are some interesting facts you may need to know:

- Some C++ users may be curious as to when virtual tables are emitted. Normally you don't have to worry about the virtual tables (*vtables*), which are generated to support virtual functions. MPW C++ usually generates the tables in just one object file (out of all the files that use the class), and the link will work correctly. There should be no duplicate or missing vtables. (Names of vtable begin with `__ptbl` and `__vtbl`.)

  You should be careful to recompile all source files that reference a class when you change the class, even if you think the change does not affect some files. If you don't recompile the file that happens to generate the vtables for the class, the tables will not be updated and your program may not run correctly.

  Sometimes the vtables for a class are generated from all source files that use the class. When this happens, you will get a "duplicate symbol" warning from the linker. You can safely ignore this warning. You can also fine-tune the vtable generation using the `-vtbl0` and `-vtbl1` options. For example, you can create a file that includes all of your class declarations, then compile that file with `-vtbl1` and all others with `-vtbl0`.

  In some rare cases (such as when a virtual function is written in assembly code), you may need to know how CFront decides which file to put the vtables into. The exact criteria are subject to change, but the current method is given below. (This description is only approximate.)

  In general, CFront tries to find a so-called key function in the class. For regular classes (not derived from PascalObject), this function is the first noninline virtual function in the class definition. (A function is considered inline if the body is given in the class, or if the definition of the function uses the inline keyword.) If there is a key function, then virtual tables will be generated for a file only if that function is defined (has a body) in the file. Since only one file should define the key function, only one copy of the vtables should be produced. If there is no key function, vtables are always generated. This may lead to duplicate vtables and warnings from the linker.

  Classes derived from PascalObject use a similar method to determine when to emit method tables. However, the criteria for selecting the `key` function are too complex to describe here.

- In order to handle static constructors and destructors in C++ code, CFront and the linker conspire to create a special segment named `%_Static_Constructor_Destructor_Pointers`.
  This segment is checked as the application starts up to see whether any static data initialization or cleanup is required. Removing this segment or changing its name will cause this process to fail.

Chapter 3 **MPW C++ Language Extensions**

This chapter describes the MPW C++ language extensions and optimizations developed by Apple Computer, Inc., for the Macintosh programming environment. The material covered in this chapter applies only to this implementation of C++. The following sections explain how to use the toolbox from C++, how to use relocatable heap objects, and how to a mixture of Object Pascal and C++ modules.

Refer to Appendix C, "Calling Conventions and Type Correspondence," in the *MPW 3.0 C Reference* for a complete description of the Pascal-style and C-style calling conventions.

Toolbox support

MPW C++ contains extensions beyond AT&T C++ in order to support the Macintosh Toolbox. The Toolbox uses Pascal-style calling conventions, which differ from C-style calling conventions. To support these differences while still allowing mixing of Pascal and C code, MPW 3.1 C++ supports both C-style calling conventions and Pascal-style calling conventions, as does MPW 3.1 C. Chapter 4, "Using the Macintosh Interface," of the *MPW 3.0 C Reference* describes the C header files in detail.

Using the type modifier `pascal`

A Pascal-style external function declaration contains the keyword `pascal`. Consider the following Pascal procedure:

```
PROCEDURE MyText(bytecount: integer; textAddr: Ptr; numer, denom: Point);
```

Because it returns no value, this procedure is of type `void` in C, so the declaration is

```
pascal void MyText (short byteCount, Ptr textAddr, Point numer,
   Point denom);
```

C and Pascal support different data types. When a C program and a Pascal program use the same global or external variables, they must use the corresponding data types. Therefore, when writing a Pascal-style function declaration in C, you must provide a translation of the parameter types and function-result type (from Pascal to C).

Often this translation is obvious, but not always. For example, Pascal passes type `char` as a 16-bit value. Table C-1 in the *MPW 3.0 C Reference* summarizes this translation.

String parameters are null-terminated C strings unless otherwise indicated. `ResTypes` and `OSTypes` can be expressed as character literals; for example, `'MENU'`. All `VAR` parameters in external Pascal declarations must be passed explicitly by pointer or reference.

Pascal-style functions may not be overloaded, because the function names do not contain the type signature used to distinguish between the versions of an overloaded function. Similarly, Pascal-style functions may not be type-conversion functions or operator functions.

MPW C and MPW C++ both treat Pascal-style functions the same way.

MPW® C++

Version 3.1

This package contains

4	Manuals	*MPW C++ Reference* *AT&T Product Reference Manual* *AT&T Library Manual* *AT&T Selected Readings*
1	Set of release notes	*MPW C++ Release Notes*
2	Disks	*MPW C++ 1,* Disk 1 of 2 *MPW C++ 2,* Disk 2 of 2
1	2-inch binder	
5	Tabs	

If you have any questions, please call

1-800-282-2732 (U.S.)
1-408-562-3910 (International)
1-800-637-0029 (Canada)

M0346LL/E

5/2/91

Interfacing to the Standard Apple Numerics Environment (SANE)

MPW C++ fully supports the Standard Apple Numerics Environment (SANE). SANE refers to numerics facilities, conforming to the IEEE Standard for Binary Floating-Point Arithmetic, uniformly available for all Apple computers. SANE is documented completely in the *Apple Numerics Manual*, second edition.

Floating-point arithmetic in MPW C++ behaves as in MPW 3.1 C. All facilities in the C SANE and Math libraries (SANE.h and Math.h) may be used in C++ programs. In particular, the data types `comp` and `long double` are supported. For compatibility, the obsolete name `extended` is allowed as a synonym for `long double`.

In order to accommodate the special IEEE Standard values NaN (not-a-number) and Infinity, MPW C++ uses the same I/O syntax for the stream library as MPW C does for `scanf` and `printf`. The stream operator >> recognizes `INF` as Infinity and the string `NAN` as a NaN. `NAN` may be followed by parentheses, which may contain an integer (a code indicating the NaN's origin). `INF` and `NAN` are optionally preceded by a sign and are case-insensitive. The stream operator << writes Infinities as `INF` and NaNs as `NAN(`*ddd*`)`, where *ddd* is the NaN code; `INF` and `NAN(`*ddd*`)` may be preceded by a sign.

Most portable C++ programs will run under MPW C++ without modification.

Refer to the *MPW 3.0 C Reference* and the *Apple Numerics Manual* for detailed information on SANE.

Support for the MC68881 and MC68882 coprocessors

The MC68881 and MC68882 math coprocessors execute basic arithmetic and a number of transcendental functions very rapidly. The Macintosh II always takes advantage of these coprocessors to speed floating-point arithmetic, even if the object code it is running was not compiled specifically for the Macintosh II. This performance gain is possible because the MPW C++ Compiler calls the SANE arithmetic routines (located in the Macintosh ROMs), which pass some calls on to the coprocessors. The ROMs in Macintosh computers without the MC68881 or Mc68882 chips use their own floating-point routines.

The MPW C++ Compiler makes direct use of these math coprocessors through the `-mc68881` command line option. This option generates MC68881 and MC68882 code for arithmetic operations and comparisons, instead of emulating them in the SANE Library. The results of all functions are the same.

The key difference between choosing the SANE Library routines (floating-point software emulation) and the `-mc68881` option (hardware implementation) for floating-point arithmetic operations is the storage size of the `long double` data format. The MC68881 extended format contains 16 bits of padding so that the data can fit evenly into three 32-bit memory accesses. The SANE extended format is 80 bits. In order to avoid the hazards of one data type with a variable size, compile all your modules by using either the SANE Library convention (80 bits) or the `-mc68881` option (96 bits).

MPW C++ provides the same support for the Motorola MC68881 (or MC68882) floating-point coprocessor as does MPW 3.1 C. Refer to the *MPW 3.0 C Reference* for more detailed information.

Six of the C libraries (CLib881.o, CPlusLib881.o, CPlusOldStreams881.o, CSANELib881.o, Math881.o, Complex881.o) are for use only with the MC68881 and MC68882 floating-point chips on the Macintosh II. (For more information, see the section "Linking a MPW C++ Program with Libraries," in Chapter 2.)

Optimized `enum` constants

In standard C++, an `enum` constant is equivalent to an `int` and would always be represented as a 32-bit quantity. For compatibility with the Macintosh Toolbox, MPW C++ optimizes storage for enumerations: that is, a variable of type `enum` that requires only a single byte of memory is allocated only one byte of storage. Similarly, an `enum` constant requiring 16 bits to represent its value is allocated a word (16 bits) of storage. An `enum` constant is allocated 32 bits of storage when its value requires it or by explicitly using the `-z6` compiler option. This option overrides the allocation algorithm discussed above and forces the compiler to allocate 32 bits (`int`) for enumerated data types.

- *Note:* Although the `-z6` compiler option conforms to the convention specified for ANSI C, it does not match the enumerated data structures used in the Macintosh ROM.

Syntax for direct function calls

MPW C++, like MPW C, allows you to make direct function calls. These are used to call subroutines in the Macintosh Toolbox or to put small pieces of assembly-language code into your program.

In calls to direct functions, one or more 680x0 instructions replace the usual subroutine call (JSR) instruction in the calling sequence. Direct functions are used to specify traps to the Macintosh ROM. Multiple 680x0 instructions may be used to specify package traps. Direct functions can also be used to access specialized machine instructions using inline code. Direct functions may have either the normal or Pascal calling conventions.

In a direct function definition, the function body is replaced by a direct function initializer. The syntax is similar to that of the initialization of variables:

direct-function-initializer:
= *instruction*
= { *instruction-list* }

instruction-list:
instruction
instruction-list, instruction

The syntax
`void Foo (int x, short y) =` *{ instruction1, instruction2,...}*

or
`void Foo (int x, short y) =` *instruction;*

where
instruction1, instruction2,...

is interpreted as a list of machine instructions specified as constants (each of which fit into 16 bits).

Each *instruction* must be in the range of either type `short` or type `unsigned short` (that is, –32,768 to 32,767 or 1 to 65,535), and is interpreted as a 16-bit 680x0 instruction. These instructions replace the JSR instruction in the calling sequence for the function. For example, the two definitions

```
pascal void PenSize (short width, short height) = 0xA89B;
pascal TEHandle TEStylNew(Rect *destRect, Rect *viewRect) = 0xA83E;
```

define Pascal-style functions that are direct traps to the Macintosh ROM.

A list of instructions, in braces, is also permitted. These two examples are multiple direct functions

```
pascal long SetCurrentA5(void) = {0x2e8d,0x2a78,0x0904};
pascal void SetStylHandle(TEStyleHandle theHandle, TEHandle hTE)
                          = {0x3F3C,0x0005,0xA83D};
```

- *Note:* Direct functions can in effect be used for limited inline machine-language programming, although this practice is not recommended for the faint of heart. When the C compiler encounters a direct function call, it is treated exactly as a function call that is, the compiler saves any values needed from the "scratch registers," pushes parameters and space for the result if necessary, and then inserts, instead of a JSR instruction, the instruction list from the direct function call. Finally, the scratch registers are restored to their previous values. (Thus a direct function can use the scratch registers without restoring them.)

Macintosh Operating System and Macintosh Toolbox functions located in ROM are called directly with A-traps. In the Motorola 680x0 CPUs, traps are software interrupts caused by instructions. In the Macintosh, trap instructions are reserved for toolbox and operating system routines.

For example, consider these samples taken from the Macintosh Toolbox interface library. In Controls.h is the direct function call

```
pascal ControlHandle GetNewControl
   (short controlID, WindowPtr owner)=0xA9BE;
```

And in Dialogs.h is the direct function call

```
pascal short GetAlrtStage(void) = {0x3EB8, 0x0A9A};
```

When a direct function is called, the compiler emits the following sequence of instructions:

- Push arguments onto the stack, as if calling a function.
- In place of a normal function call, execute the indicated instructions as code.
- Adjust the stack pointer as if an actual function or procedure had been called.

This syntax can be viewed as access to machine-level conventions, similar to the inline assembly-language construct used with other compilers. Because of the inline nature of direct function calls, some restrictions apply:

- The unary & (address-of) operator cannot be applied to direct functions. Thus it is impossible to create pointers to direct functions or to pass direct functions as parameters.
- Virtual functions cannot be direct functions.

Using Pascal strings in C++

In general, whenever a C program handles an *n*-byte string like

```
"hello, world"
```

it stores it as an array of *n*+1 characters whose last element is the null byte (`\0`). The Macintosh ROM, however, expects Pascal strings, which have an initial length byte followed by *n* string characters (and no null byte at the end). Because both string formats have an extra byte of information (either a length at the beginning or a null byte at the end), you can transform a string in place from Pascal to C format and vice versa. The routines `c2pstr()` and `p2cstr()` in the library CInterface.o perform these conversions. (Refer to Chapter 4, "Using the Macintosh Interfaces," in the *MPW 3.0 C Reference.*)

If you want to create your own Pascal string, simply prefix a byte count to your string, in one of these two ways:

```
unsigned char *pascalstring = "\pHello";

char *pascalstring = "\005Hello";
```

The `\p` causes the string to be prefaced with a correct byte count. Both examples put an extra null byte at the end and have the following properties:

- `&pascalstring[0]` is a pointer to a Pascal string.
- `&pascalstring[1]` is a pointer to a C string.

C++ strings have the type array of `char`.

Pascal strings beginning with `\p` have the type array of `unsigned char`.

Normally, you wouldn't use Pascal strings in C++, because they are limited to 255 characters in length and C strings are limited only by memory. You use Pascal strings in C++ only when you need to use Pascal routines that expect them or return them.

Built-in classes

MPW C++ provides three specialized abstract classes. `SingleObject` improves performance when multiple inheritance is not needed. `HandleObject` and `PascalObject` facilitate Macintosh memory management.

MPW C++ memory models

Normal C++ objects can be of three kinds:

1. Global (static) objects, which are declared with the keyword `static` or declared outside any function.
2. Stack objects, which are local variables declared within a function, except those declared `static`.
3. Objects allocated from the free store, that is, objects referenced by a pointer returned by the `new` operator (also known as "dynamically allocated" objects, or "objects on the heap").

MPW C++ allows a fourth kind of object—handle-based dynamic objects—which are declared and used exactly as pointer-based objects, but implemented using handles instead of pointers. Such objects can be only in the heap, although their handles can be located elsewhere.

Handle-based objects are an MPW C++ extension to standard C++. They are included to simplify the use of the Macintosh Toolbox Memory Manager for allocating class objects. A special type of handle-based object, the `PascalObject`, lets you declare classes of objects identical to objects generated by Object Pascal. "Class `PascalObject` Handle-Based Classes," later in this chapter, describes these classes. For more details on memory management in the Macintosh and a discussion of handles, consult *Inside Macintosh,* Volume I.

A `HandleObject` is similar to other C++ objects, but it is accessed through a handle; that is, a pointer to a master pointer, as returned by the Memory Manager. The predefined class `HandleObject` allows you to make direct use of the Memory Manager.

A class is defined as handle-based if it is derived from one of the base classes `HandleObject` and `PascalObject`, or from another handle-based class. Thus, there are two hierarchies of handle-based classes. The root of one hierarchy is the predefined class `HandleObject`; the root of the other is the predefined class `PascalObject`.

Class `SingleObject`: single-inheritance hierarchies

Virtual functions are implemented using a data structure called a *virtual table.* Virtual tables that support multiple inheritance are twice as large as those that only support single inheritance. The overhead for calling virtual functions is also higher when multiple inheritance is supported.

Not all applications need multiple inheritance. As an extension to standard C++, MPW C++ supports a single-inheritance hierarchy. All classes derived from `SingleObject`, whether directly or indirectly, are restricted to using single inheritance. The virtual tables for these classes are much smaller, and virtual function calls are somewhat faster.

`SingleObject` classes are provided only for efficiency. Their use should be limited to those classes where limiting virtual-table size or virtual-function-call overhead is important.

Class `HandleObject`: C++ handle-based classes

Objects of handle-based classes can be accessed only through variables declared as pointers. MPW C++ treats these pointers as handles. Here is an example:

```
class MyObjects : HandleObject {
// field and member function definitions
}

MyObjects * myObj = new MyObject;
```

Implementation of C++ handle-based classes

Handle-based classes are implemented by using the handles returned by the Macintosh Toolbox Memory Manager, and not pointers. Therefore, each dereference of a pointer to an indirect class is actually a double dereference: one to get the actual pointer stored at the location specified by the handle, and one to access the data in the object.

The default `new` and `delete` operators are implemented differently for handle-based classes, as they allocate and free memory by using handles instead of pointers. `HandleObject` classes (non-Pascal handle-based classes) that contain virtual member functions use standard C++ virtual function calling conventions; `PascalObject` classes (Pascal classes) use the Object Pascal conventions described later in this chapter.

Restrictions on all handle-based classes

Because handle-based classes can only be accessed through handles, they are restricted in certain ways. This section describes restrictions that apply to *all* handle-based classes, including Pascal classes. For additonal restrictions that apply to Pascal classes *only,* see "Restrictions on Pascal Handle-Based Classes," later in this chapter.

- Multiple inheritance cannot be used with handle-based classes.

- Handle-based objects can be created only by the `new` operator. The only legal use of a dereferenced handle-based class pointer (for example, `*x`) is to refer to a field in the class (for example, `*x.y` or `x->y`).
- Pointers to handle-based classes cannot be cast to any other type except to a pointer to another handle-based class. Pointers to anything else cannot be cast to a pointer to a handle-based class.
- No address arithmetic can be performed on a pointer to a handle-based class, except the implicit arithmetic used in a member reference.
- It is legal (but unsafe, because the object may move) to take a pointer to a field of an indirect class (for example, `&x->y`). This restriction includes the implicit use of pointers by references, such as `int& p = x->y`. In a future release, the compiler may give errors for these unsafe references, and there may be a compiler option and a pragma to disable individual errors.
- Because handle-based objects must exist on the Macintosh heap, it is an error to declare global variables, local variables, arrays, members, or parameters of handle-based types (rather than pointers to them).

Arrays of handle-based classes

In C++, if you can allocate a single `MyObject` by saying

```
MyObject * p = new MyObject;
```

Then you can allocate an array of type `MyObject` by saying

```
MyObject * mp [] = new MyObject[10];
```

which allocates and constructs 10 of them. If `MyObject` is handle-based, this does not work.

The workaround for this is to write a loop:

```
typedef MyObject * MyObjectsPtr;
MyObjectsPtr * mp_array = new MyObjectsPtr [10];  // allocate space for
the pointers
for (int i = 0; i < 10;  i++)
   mp_array[i] = new MyObject;                    // allocate the
objects
```

Class `PascalObject`: Pascal handle-based classes

Object Pascal consists of object-oriented extensions to Pascal that resemble C++, Smalltalk-80™, and Simula-67. Object definitions are similar to Pascal records, but contain both data fields and functions. The member functions in Object Pascal are called *methods.* (For more information on Object Pascal, refer to the *MPW 3.1 Pascal Reference.*)

Pascal handle-based classes are derived from the predefined class `PascalObject`. Pascal classes make it possible for you to use Object Pascal and MacApp, the extensible Macintosh application written in Object Pascal, from MPW C++ programs. MacApp is a skeleton application; it defines and implements a hierarchy of Object Pascal classes that lets you easily create an application with the standard Macintosh user interface. For more information about the rich and diverse functionality in the MacApp libraries, refer to the *MacApp 2.0 Reference.*

A key concept in object-oriented programming is run-time binding of function calls. MPW C++ and MPW Pascal use radically different techniques to call methods dynamically. To accommodate the mixing of Object Pascal and MPW C++, MPW C++ uses the Pascal method-dispatching scheme whenever a virtual function is a member of a class derived from the built-in class `PascalObject`. Objects of Pascal classes and objects defined in Object Pascal can be used interchangeably by functions written in either language. Objects created by Pascal programs can be passed to MPW C++ programs and used as instances of Pascal classes, and vice versa.

The virtual member functions of Pascal classes are analogous to Object Pascal methods. You can freely mix MPW C++ functions and Pascal procedures. A Pascal class in MPW C++ can use an Object Pascal method as one of its virtual member functions; an Object Pascal class can use an MPW C++ virtual member function as one of its methods. Pascal class hierarchies can be implemented in a mixture of Object Pascal and MPW C++.

As with other functions, remember that the calling conventions (C or Pascal) must match; virtual functions to be defined or referenced from Pascal code should be declared with the `pascal` keyword.

It is possible to declare a Pascal class member function `virtual` without declaring it `pascal`. It would use the Object Pascal function dispatching, but would not be accessible from Pascal code because of the differences in the calling conventions.

A Pascal class can contain nonvirtual member functions, but they do not use the Object Pascal function dispatching, and are not accessible from Object Pascal.

- *Note:* You need to declare the member functions `virtual` and `pascal`, *and* the class must be derived from `PascalObject` to be compatible with Object Pascal and MacApp.

To use an existing Object Pascal module (from the MacApp library, for example), you need a C++ header file that corresponds to the class and method definitions of the Pascal module. All of your Pascal base classes must be derived (directly or indirectly) from the predefined class `PascalObject`, and they must follow the rules given earlier for translating Pascal parameters and variables to C parameters and variables.

Implementation of Pascal handle-based classes

An Object Pascal run-time routine is used to get and free handles for Pascal classes. Virtual member functions of Pascal classes are accessed through method tables, not virtual tables. These tables are identical to Object Pascal method tables. Because no virtual tables are generated, Pascal handle-based classes do not contain a virtual table pointer. Instead, they contain a pointer to the "class info" block, which includes the method table. Member functions of Pascal handle-based classes have Pascal-type external names. For example, the member function

```
pascal void aClass::aMember()
```

is passed to the linker as `ACLASS_AMEMBER` to agree with the Object Pascal convention. If the `pascal` keyword is left out, the name `aClass_aMember` is used. Such a function is not accessible from Pascal code, but otherwise uses the Object Pascal implementation.

In the object code, when a virtual member function is called (regardless of whether the class is handle-based), the `this` parameter is always pushed on the stack last. For C++ calling conventions, this is implemented by putting `this` as the first parameter. For Pascal calling conventions, this is done by putting `this` (known as `SELF` in Object Pascal) last.

Restrictions on Pascal handle-based classes

In addition to the restrictions noted for all handle-based classes, Pascal handle-based classes have the following restrictions.

- Nonvirtual member functions cannot be declared in or called from Pascal code. Member functions can be declared as either Pascal or non-Pascal; however, non-Pascal member functions can only be correctly called from MPW C++ code.

- Constructors and destructors are allowed. As with all classes, if the class has virtual member functions, then the destructor should be virtual. With MacApp, the use of constructors should be limited to calling the conventional initialization routine. Because you can write a conventional cleanup routine to call from Pascal, you can call it in the destructor. You should not call the `FREE` method of your object from within your destructor, because it will attempt to delete the object a second time.
- Overloading, type conversion functions, and operator functions require type signatures. Because the Pascal naming conventions do not include type information, overloading, type conversion, and operator functions are not allowed for virtual members of Pascal classes, or any function with the `pascal` attribute. See *AT&T Slected Readings,* "Type-Safe Linkage for C++," for more information.
- Although operator functions, type conversion functions, and overloading are allowed for member functions of Pascal handle-based classes, they cannot be declared or accessed from Object Pascal.
- Pointers to Pascal handle-based classes cannot be cast to a pointer to a non-Pascal handle-based class, and vice versa.
- Although `new` and `delete` may be overridden for Pascal classes, the overridden functions do not have the same arguments as `new` and `delete` for other classes. Pointers are of type `void**`, not `void*`; and `new` has an additional (leading) parameter of type `pascal void (*)()`.

The keyword `inherited`

The keyword `inherited` is used for accessing a base class's version of a particular member function, without having to explicitly name the base class. For example,
`inherited::`*functionname*

refers to the instance of *functionname* that would have been found if *functionname* had not been declared in the current class.

Appendix A **MPW C++ Compiler Options**

This appendix describes the options you can use with the CPlus command, which invokes the MPW C++ compiler, and CFront, which is called by CPlus.

CFront—C++-to-C translator script; CPlus—C++-to-C translator

Syntax

`CFront` [*option* ...] [*file* ...]< *file* >[*intermediate output*]≥ [*progress*]

`CPlus` [*option* ...] [*file* ...]< *file* >[*intermediate output*]≥ [*progress*]

Description

The MPW tool CFront is a C++ translator with a built-in C preprocessor. The MPW script CPlus calls CFront (which compiles the specified C++ source file to intermediate C source code), then calls the MPW 3.1 C compiler and pipes the intermediate C code to it. (Optionally, the intermediate C code can be saved as a file.) Compiling file *Name*`.cp` creates object file *Name*`.cp.o`. (By convention, C++ source filenames end in a `.cp` suffix.) If no filenames are specified, standard input is compiled and the object file `cp.o` is created.

Normally, you will only use CPlus without the MPW C Compiler when using the options **`-c`**, **`-comp`**, or **`-patch`**: that is, if you are using CFront as the front end for some other C compiler or if you want to look at the intermediate C code generated by CFront.

All CPlus options are passed on to CFront, except **`-b`**, **`-b2`**, and **`-b3`**. These all regulate the behavior of the C compiler.

Type

Script.

Input

If no filenames are specified, standard input is compiled. You can terminate input by pressing Command-Enter.

Output

If you specify the **-e** or **-e2** option, preprocessor output is written to standard output; no compilation or syntax checking is done, and no object file is produced. If you specify the **-c** option, the C code produced by CFront is written to standard output, and the C compiler is not invoked.

Diagnostics

Errors and warnings are written to diagnostic output. If the **-p** option is specified, progress and summary information is also written to diagnostic output.

Status

The following status values may be returned:

0 Successful completion.

1 Errors occurred.

Options

-32

Passed unchanged to C compiler.

-a | -a1

Generate intermediate C code with ANSI prototypes (the default).

-a0

Do not generate ANSI prototypes. This is useful for producing C code that will be compiled by an older C compiler.

-b

Generate PC-relative references for functions in the same segment and for string constants (which are kept at the end of the function code module). The default is to place string constants in the global data area, and to generate A5-relative (jump table) references for function addresses. Useful for writing desk accessories, WDEFs, and so on. *(CPlus only.)*

-b2

Provide the actions of option **-b**, but also allow the code generator to reduce code size by overlaying the storage constants where possible. *(CPlus only.)*

-b3

Cause the code generator to keep string constants within the code and overlay them when possible (but always generate A5 relative references for function addresses). *(CPlus only.)*

-c

Do not generate object code. Write the intermediate C code to standard output. This option is useful in two circumstances: to pipe the output of CFront to a different C compiler; or to produce the C code for human inspection in order to clarify the semantics of a C++ construct.

Certain constructs, such as static constructors and `PascalObject` classes, cannot be fully represented in C. When these constructs are used, the C code alone will not work correctly in the Macintosh environment.

The C code generated with the **-c** option is a fair representation of the program if

- static (or global) objects are not being constructed or destructed
- nothing is derived from PascalObject (that is, no MacApp)

-comp name options

Causes the compiler *name* to be invoked rather than the MPW C compiler. The options, and the intermediate C file, are passed to *name*. The CPlus script must be changed as follows:

The `{®1}` command in the script will set the MPW shell variable CPlusObjName to the name of the object file that should be produced. A command must be added to rename or move the object file produced by *name* to `{CPlusObjName}`. (The name of the intermediate C file passed to *name* is `C.pipe.code.c`, and it is in the directory specified by the **-y** option.) If *name* is not being invoked (for example, if **-c** is specified before `-comp`), then `CPlusObjName` is set to the null string.

-d *name*

Define *name* to the preprocessor with value 1. This is the same as writing

`#define` *name* `1`

at the beginning of the source file. (The **-d** option does not override `#define` statements in the source file.)

-d *name=string*

Define *name* to the preprocessor with value *string*. This is the same as writing

`#define` *name string*

at the beginning of the source file.

`-dump` *filename*

Dump saved C++ compilation state to *filename*.

`-e`

Do not compile the program. Instead, write the output of the preprocessor to standard output. This option is useful for debugging preprocessor macros.

`-e2`

Implies **`-e`**, but also suppresses comments.

`-elems881`

Use inline MC68881 instructions for all transcendental functions available on the MC68881 processor. See the *MPW 3.0 C Reference* for a complete list of these functions. This option implies the **`-mc68881`** option.

`-f` *filename*

When CFront gets its source from standard input (for example, when the source is sent to a stand-alone preprocessor whose output is piped to CFront), the option **`-f`** *filename* will cause the correct line-number information to be sent to the intermediate C code. This option is only useful when using a non-MPW C compiler: it will not work correctly with SADE®.

`-fx`

Passed unchanged to C compiler.

`-h`

Passed unchanged to C compiler.

`-i` *pathname*[, *pathname*]...

Search for include files in the specified directories. Multiple `-i` options may be specified. A maximum of 15 directories can be searched. These are the rules for searching: if a partial pathname has been specified in the `#include` directive (no colons in the name or a leading colon), then directories are searched in the following order:

1. The directory containing the source file if the `#include` directive has been written `#include "..."` (the alternative is `#include <...>`, in which case this step is skipped).
2. The directories specified in the `-i` options, in the order listed.
3. The directories specified in the shell variable {CIncludes}.

`-k`

Passed unchanged to C compiler.

`-l`

In the C output file, generate `#line` *nn* directives as `#`*nn*. (This option is spelled el.)

`-l0`

Do not generate `#line` *nn* directives. (This option is spelled el zero.)

`-load`*filename*

Load saved C++ compilation state from *filename*.

`-m`

Generate 32-bit references for data. Required when there are more than 32 KB of global data.

`-mark` *fcts*| *types*| *data*

Create MPW markers for the specified items in file being compiled.

`-mark all`

Create MPW markers for all of the above.

`-maxerrors` *number*

Specify that CFront abort after reporting *number* errors (default 12).

-mbg ch8

Include version 2.0-compatible MacsBug symbols (eight characters only, in a special format).

-mbg full | on

Include full (untruncated) symbols for MacsBug.

-mbg off

Do not include symbols for the MacsBug debugger.

-mbg *number*

Include MacsBug symbols truncated to length *number*. (Does not apply to method tables for Object Pascal classes.)

-mc68020

Generate MC68020 instructions whenever doing so would provide faster and/or smaller object code.

-mc68881

Generate MC68881 instructions for all basic floating-point operations. Predefine the preprocessor symbol `mc68881` to the value 1.

-mf

Allow MPW C++ to request MultiFinder memory when the MPW shell partition is not large enough to complete the compilation. Memory is released after compilation.

-mtbl0

Suppress output of method tables for each Object Pascal class. (The last two characters are el and zero.)

-mtbl1

Force output of method tables for each Object Pascal class. By default, CFront outputs the method table as necessary. Use the **-mtbl1** option for special cases. (The last two characters are el and one.)

If neither **-mtbl0** nor **-mtbl1** is selected, the default is to attempt to emit method tables only when a key method (such as the first method in the class) is defined. If no key method can be identified, the method tables are always emitted.

-n

Turn pointer assignment incompatibility errors into warnings. (Passed unchanged to C compiler.)

-o *objname*

Pathname for the generated object file. If *objname* ends with a colon, it indicates a directory for the output file, whose name is then formed by the normal rules (that is, *inputfilename*.o). If *objname* does not end with a colon, the object file is written to the file *objname*.

-p

Write progress information (include file names, function names, and sizes) and summary information (number of errors and warnings, code size, global data size, and compilation time) to diagnostic output. Some of this information is generated by CFront, and some by C.

-patch

Generate intermediate C code suitable for a system that uses the AT&T "patch" mechanism for static constructors. Should only be used with the **-comp** option.

-q

Passed unchanged to C compiler.

-s *segment*

Name the object code segment. (The default segment name is Main.)

-sym off

Do not generate SADE records.

-sym on | full

Write complete object file records containing information for SADE, the MPW symbolic debugger. This option can be limited by also specifying one or more of `nolines`, `novars`, `notypes`—which cause omission of line, type, or variable information, respectively, from the object file (for example, `-sym on, nolines, novars`). For more information, see the *MPW 3.0 C Reference*.

-t

Write C compilation time to diagnostic output. (Passed unchanged to C compiler.)

-trace

Passed unchanged to C compiler.

-u *name*

Undefine the predefined preprocessor symbol *name*. This is the same as writing

#undef *name*

at the beginning of the source file.

-v

Passed unchanged to C compiler.

-vtbl0

Suppress output of virtual tables for each ordinary class with virtual functions. (The last two characters are el and zero.)

-vtbl1

Force output of virtual tables for each ordinary class with virtual functions. By default, CFront outputs the virtual table as necessary. Use the `vtbl1` option for special cases. (The last two characters are el and one.)

If neither **-vtbl0** nor **-vtbl1** is selected, the default is to attempt to emit virtual tables only when a key function (such as the first virtual noninline function in the class) is defined. If no key function can be identified, the virtual tables are always emitted.

-w

Suppress compiler warning messages. (By default, warnings are written to diagnostic output.)

-w1

Cause CFront to generate additional warnings.

-w2

Cause CFront and the C compiler to generate all possible warnings.

-w3

Cause C compiler to suppress "unused" warnings.

-x *filename*

Generate intermediate C code for a different processor, using the size and alignment information in *filename*.

-y *pathname*

Put the CFront temporary files used by CPlus (C.pipe.) and the C compiler's temporary intermediate files (≈.o.i) in the directory specified by *pathname*.

If the **-y** option is not given, the default is to use the directory specified in the exported shell variable {CPlusScratch}; if that variable does not exist, {MPW} is used; if that does not exist, the current directory is used. The CPlus script normally deletes the scratch files, but terminating the script with Command-period or invoking CFront directly will cause the files to remain.

-z 0

Force inline functions to be noninline.

-z 3

Normally, CFront produces intermediate C code using variable names identical to those in the C++ source whenever possible. This makes debugging with SADE much easier. Sometimes, such as when the global scope operator (::) is used, it is necessary to encode a name so that the C compiler can distinguish two uses of the same name.

AT&T's CFront always encodes local names and members. MPW's CFront avoids encoding whenever practical. The **-z3** option guarantees this condition.

-z 4

Normally, CFront produces intermediate C code using variable names identical to those in the C++ source whenever possible. This makes debugging with SADE much easier. Sometimes, such as when the global scope operator (::) is used, it is necessary to encode a name so that the C compiler can distinguish two uses of the same name.

AT&T's CFront always encodes local names and members. MPW's CFront avoids encoding whenever practical. It is possible (though no known cases exist) that the MPW CFront may sometimes remove a necessary encoding. If this happens, the MPW C compiler will report an error in the intermediate C code. This problem could be corrected by specifying the **-z4** option, which restores full encoding.

-z 6

Do not optimize enumerations: `enum` variables become the same as `int`.

-z 7

Relax the requirement on static class member initialization.

Example

```
cplus -p Sample.cp
```

Compiles Sample.cp, producing the object file Sample.cp.o. Writes progress information to diagnostic output.

See also *MPW 3.0 C Reference.*

Appendix B **MPW C++ Keywords**

The following identifiers are reserved for use as keywords, and may not be used for other purposes. MPW C++ keywords are in normal type. Apple extensions are underlined.

MPW C++ keywords are a combination of C++ keywords, as defined in the *UNIX System V AT&T C++ Language System Release 2.0: Product Reference Manual,* and MPW 3.1 C keywords. The Apple keywords `comp`, `extended`, `inherited`, and `pascal` are defined in Chapter 3, "MPW C++ Language Extensions," and in the *MPW 3.0 C Reference.* The keywords `catch`, `template`, and `volatile` are reserved for future language extensions.

`asm`	`extern`	`register`
`auto`	`float`	`return`
`break`	`for`	`short`
`case`	`friend`	`signed`
`catch`	`goto`	`sizeof`
`char`	`if`	`static`
`class`	`inherited`	`struct`
`comp`	`inline`	`switch`
`const`	`int`	`template`
`continue`	`long`	`this`
`default`	`nev`	`typedef`
`delete`	`operator`	`union`
`do`	`overload`	`unsigned`
`double`	`pascal`	`virtual`
`else`	`private`	`void`
`enum`	`protected`	`volatile`
`extended`	`public`	`while`

Appendix C MPW C++ Complex Mathematics Library

The Complex Mathematics library supplied by MPW C++ is a superset of that provided by AT&T C++: it has been extended to provide complete support for IEEE-standard binary floating-point arithmetic.

A general description of the AT&T library is in Chapter 1 of the *UNIX System V AT&T C++ Translator Release 2.0: Library Manual.* These manual pages replace those in that chapter to describe the changes necessary for MPW C++ in order to support the SANE `extended` type.

Complex introduction

Definition Introduction to MPW C++ Complex Mathematics library

Synopsis

```
#include <complex.h>
class complex;
```

Description This section describes functions and operators found in the C++ Complex Mathematics library. Declarations for these functions may be found in the header file complex.h.

The Complex Mathematics library implements the data type of complex numbers as a class, `complex`. It overloads the standard input, output, arithmetic, assignment, and comparison operators, discussed in the manual pages for operators. It also overloads the standard exponential, logarithm, power, and square root functions, discussed in "exp" later in this appendix , and the trigonometric functions of sine, cosine, hyperbolic sine, and hyperbolic cosine, discussed in "trig" later in this appendix, for the class `complex`. Routines for converting between Cartesian and polar coordinate systems are discussed in "Cartesian/polar" later in this appendix. Overloaded insertion and extraction operators are also available if either iostream.h or stream.h is included before including complex.h.

The routines in the MPW++Complex Mathematics library, based upon algorithms by W. Kahan, have been carefully crafted to exploit the features of the IEEE conforming arithmetic provided by SANE. Even beyond the natural advantages provided by the use of the `extended` data type, the algorithms are designed to maximize precision and to avoid premature overflow or underflow. In addition, careful attention has been paid to the treatment of NaNs and Infinities. The importance of a signed 0 at complex branch cuts is evident for several inverse functions. For example, the complex square root of an argument lying on the negative real axis depends upon the sign of the 0 of its imaginary component: (sqrt(–1 + 0i) = +i; sqrt(–1 – 0i) = –i).

Files complex.h

complex.o

complex881.o

Stream libraries as required; see Appendix D.

See also Cartesian/polar, Operators, `exp`, and `trig`.

Diagnostics Functions in the Complex Mathematics library follow the SANE conventions with respect to exceptions.

Cartesian/polar

Definition Cartesian/polar: `abs`, `arg`, `conj`, `imag`, `norm`, `polar`, `real`—for the C++ Complex Mathematics library

Synopsis

```
#include <complex.h>>

class complex {

public:
          friend extended abs(complex);
          friend extended arg(complex);
          friend complex conj(complex)
          friend extended imag(complex);
          friend extended norm(complex);
          friend complex polar(extended, extended = 0);
          friend extended real(complex);  };
```

Description The following functions are defined for `complex`, where `d`, `m`, and `a` are of type `extended`, and `x` and `y` are of type `complex`.

`d = abs(x)`	Returns the absolute value or magnitude of `x`.
`d = norm(x)`	Returns the square of the magnitude of `x`. It is faster than `abs`, but more likely to cause an overflow error. It is intended for comparison of magnitudes.
`d = arg(x)`	Returns the angle of `x`, measured in radians in the range $-\pi$ to π, inclusive.
`y = conj(x)`	Returns the conjugate of `x`. That is, if `x` is `(real, imag)`, then `conj(x)` is `(real, -imag)`.
`y = polar(m, a)`	Creates a complex given a pair of polar coordinates, magnitude `m`, and angle `a`, measured in radians.
`d = real(x)`	Returns the real part of `x`.
`d = imag(x)`	Returns the imaginary part of `x`.

See also Complex introduction, Operators, `exp`, and `trig`.

Operators

Definition complex_operators: operators for the C++ Complex Mathematics library

Synopsis

```
#include <complex.h>
class complex {

public:
      friend complex operator+(complex, complex);
      friend complex operator-(complex);
      friend complex operator-(complex, complex);
      friend complex operator*(complex, complex);
      friend complex operator*(complex, extended);
      friend complex operator*(extended, complex);
      friend complex operator/(complex, complex);
      friend complex operator/(complex, extended);
      friend complex operator/(extended, complex);

      friend int operator==(complex, complex);
      friend int operator!=(complex, complex);

void operator+=(complex);
void operator-=(complex);
void operator*=(complex);
void operator*=(extended);
void operator/=(complex);
void operator/=(extended);
};
```

Description The basic arithmetic operators, comparison operators, and assignment operators are overloaded for complex numbers. The operators have their conventional precedences. In the following descriptions for complex operators, `x`, `y`, and `z` are of type `complex`. Note: for efficiency reasons, special cases are defined for the `*` and `/` operators when one of the operands is of type `extended`, and for the `*=` and `/=` operators when the right-hand operand is of type `extended`.

Arithmetic operators

```
z = x + y
```

Returns a complex which is the arithmetic sum of complex numbers `x` and `y`.

```
z = -x
```

Returns a complex which is the arithmetic negation of complex number `x`.

```
z = x - y
```

Returns a complex which is the arithmetic difference of complex numbers `x` and `y`.

```
z = x * y
```

Returns a complex which is the arithmetic product of complex numbers `x` and `y`.

```
z = x / y
```

Returns a complex which is the arithmetic quotient of complex numbers `x` and `y`.

Comparison operators

```
x == y
```

Returns nonzero if complex number `x` is equal to complex number `y`; returns zero otherwise.

```
x != y
```

Returns nonzero if complex number `x` is not equal to complex number `y`; returns zero otherwise.

Assignment operators

```
x += y
```

Complex number `x` is assigned the value of the arithmetic sum of itself and complex number `y`.

```
x -= y
```

Complex number `x` is assigned the value of the arithmetic difference of itself and complex number `y`.

```
x *= y
```

Complex number `x` is assigned the value of the arithmetic product of itself and complex number `y`.

```
x /= y
```

Complex number `x` is assigned the value of the arithmetic quotient of itself and complex number `y`.

Warning The assignment operators do not produce a value that can be used in an expression. That is, the following construction is syntactically invalid:

```
complex x, y, z;
      x = ( y += z );
```

whereas

```
      x = ( y + z );
      x = ( y == z );
```

are valid.

See also Complex introduction, Cartesian/polar, `exp`, and `trig`.

exp

Definition `exp`, `log`, `pow`, `sqrt`: exponential, logarithm, power, square root functions for the MPW C++ Complex Mathematics library

Synopsis

```
#include <complex.h>

class complex {

public:
      friend complex exp(complex);
      friend complex log(complex);
      friend complex pow(extended, complex);
      friend complex pow(complex, long);
      friend complex pow(complex, extended);
      friend complex pow(complex, complex);
      friend complex sqrt(complex);
      friend complex sqr(complex);
};
```

Description The following math functions are overloaded by the MPW C++ Complex Mathematics library, where - `x`, `y`, and `z` are of type `complex`.

`z = exp(x)` Returns e^x.

`z = log(x)` Returns the natural logarithm of `x`.

`z = pow(x,y)` Returns x^y.

`z = sqrt(x)` Returns the square root of `x`, contained in the first or fourth quadrants of the complex plane.

`z = sqr(x)` Returns the square of `x`.

See also Complex introduction, Cartesian/polar, Operators, and `trig`.

trig

Definition trig: sin, cos, sinh, cosh: trigonometric and hyperbolic functions

Synopsis

```
#include <complex.h>

class complex {

public:
      friend complex sin(complex);
      friend complex cos(complex);
      friend complex tan(complex);
      friend complex sinh(complex);
      friend complex cosh(complex);
      friend complex tanh(complex);
      friend complex asin(complex);
      friend complex acos(complex);
      friend complex atan(complex);
      friend complex asinh(complex);
      friend complex acosh(complex);
      friend complex atanh(complex);
};
```

Description The following trigonometric functions are defined for the Complex Mathematics library, where `x` and `y` are of type `complex`:

`y = sin(x)`	Returns the sine of `x`.
`y = cos(x)`	Returns the cosine of `x`.
`y = tan(x)`	Returns the tangent of `x`.
`y = sinh(x)`	Returns the hyperbolic sine of `x`.
`y = cosh(x)`	Returns the hyperbolic cosine of `x`.
`y = tanh(x)`	Returns the hyperbolic tangent of `x`.
`y = asin(x)`	Returns the inverse sine of `x`.
`y = acos(x)`	Returns the inverse cosine of `x`.
`y = atan(x)`	Returns the inverse tangent of `x`.
`y = asinh(x)`	Returns the inverse hyperbolic sine of `x`.
`y = acosh(x)`	Returns the inverse hyperbolic cosine of `x`.
`y = atanh(x)`	Returns the inverse hyperbolic tangent of `x`.

See also Complex introduction, Cartesian/polar, Operators, and `exp`.

Appendix D **MPW C++ Stream Library**

The AT&T CFront 2.0 Stream library is fully described in Appendix A of the *UNIX System V AT&T C++ Translator Release 2.0: Library Manual.* The MPW C++ version of this library is nearly identical to the AT&T version, save that a few lines have been added to the interface to support SANE types. The AT&T C++ Release 1.2 implementation of the Stream library is provided for backward compatibility, but is not supported. (On our system we have renamed the AT&T file OStream.h to OldStream.h.)

In order to support the SANE type `comp` and the ANSI type `long double` (equivalent to the SANE type `extended`), the following publicly accessible member functions were added to the `istream` and `ostream` classes:

Added to class `istream` (Appendix A-4, ISTREAM(3C++), page 1):

```
istream&   operator>>(comp&);
istream&   operator>>(extended&);
```

Added to class `ostream` (Appendix A-4, OSTREAM(3C++), page 1):

```
ostream&   operator<<(extended);
```

The corresponding output routine for the `comp` type is not strictly necessary since arguments of type `comp` are passed as `long double`.

Appendix E **MPW C++ Style Guide**

This style guide was originally written as a guide to C++ programming at Apple; it has been revised for more general use.

Source file conventions

The following conventions will keep your source files easy to read, easy to use, and legally protected.

Include a copyright notice

In order to protect your intellectual property rights, include the following line at the front of *every* file you create:

```
// Copyright © 1990 Your name or company.  All rights reserved.
```

You can make the © by typing Option-g. The "All rights reserved" is specifically for our foreign friends: it protects copyrights in Berne Convention countries. In addition, any binary files you ship should contain a copyright notice somewhere.

If you modify a file in more than one calendar year, you must list every year in which you modified it, for example,

```
// Copyright © 1986, 1988-1990  Your name or company.  All rights reserved.
```

Add helpful comments

Comments are helpful for those who must read or maintain your code. Comments that explain subtleties in the code are more helpful than comments like `assign a to b`.

Be careful about omitting argument names in function prototypes

You can omit dummy argument names in function prototypes, but only if the meaning is clear without them. You generally need to include argument names when you have more than one argument of the same type, as it's impossible to figure out which one is which otherwise.

If you are getting compiler warnings of the form "`warning: foo not used`" where `foo` is an argument to a function, leaving the argument name out of the function header for the function's *definition* will stop the warning. Whether or not an argument name appears in the function's *declaration* has no bearing on the warning.

Put only related classes in one file

In order to keep your class definitions under control and to make life easier for those trying to decipher them, follow the lead of the MPW {CIncludes} files. Limit each header file to a single class definition or a set of related class definitions. MPW C has always followed this convention: for example, Windows.h,Controls.h instead of Toolbox.h.

On the implementation side, put only one class implementation in a given source file (classes private to the implementation of the class may be declared and implemented in the same source file). Name the file after the class, but without the initial *T*: for example, put the class TView in ViewView.cp.

Make it easy to use your header files

Trying to figure out whether you've included all the necessary antecedents for a header file is difficult. To save your clients this trouble, enclose the definitions in your header with code that looks like the following:

```
#ifndef __MYCLASS__
#define __MYCLASS__
#include "prerequisite1.h"
#include "prerequisite2.h"
... definitions for MyClass
#endif
```

Now you can include your header's prerequisites without caring whether they've already been included elsewhere (assuming that everyone follows this convention). The name of the preprocessor variable should be all uppercase and consist of the filename (without .h) surrounded by two underlines on either side.

To speed up compilation, you can even do this in your files that use foo.h:

```
#ifndef __FOO__
#include "foo.h"
#endif
```

These lines avoid the overhead of reading and parsing foo.h.

Store files in Projector

As soon as a file is published for use by others (for example, you put it on a file server so others can use it), you should start storing it in Projector. This lets you recreate old versions if necessary and makes sure things don't get lost.

Since the whole point of using Projector is to make it easier for those who follow you in the great chain of software being, please use the features that will make their lives easier. Try to maintain a proper set of versions (for example, don't remove your file from Projector then add it again—that loses all the revisions), and use the comment features when you check things in and out.

Naming conventions

In order to make C++ even more readable, you should adopt a consistent set of naming conventions. Apple programmers use the following conventions.

Type names

All type names begin with a capital letter. In addition, class names begin with a *T* for base classes, and *M* for mix-in classes. (See "Multiple Inheritance," later in this Appendix.) Examples: Boolean, TView, MAdjustable. Never use C types directly.

Member names

Member names should begin with an *f*, for "field." Member function names need only begin with a capital letter. Examples: fVisible, Draw.

Global names

Names of global variables (**including** static members of classes) should begin with a *g*. Examples: gApplication, TGame::gPlayingField.

Local and parameter names

Names of local variables and function arguments should begin with a word whose initial letter is lowercase. Examples: i, thePort, aRegion.

Constant names

Names of constants should begin with a k. Example: kSaveDialogResID.

Abbreviations

It's best to avoid abbreviations, especially *ad hoc* ones. Inconsistent use of abbreviations makes it hard for clients to remember the correct name of a function or variable. Abbreviations are okay as long as they are consistent and universal. For example, don't use VisibleRegion some places and VisRgn others; use one or the other throughout.

Multiple-word names

In any name that contains more than one word, the first word should follow the convention for the type of the name, and subsequent words should immediately follow, with the first letter of each word capitalized. Do not use underscores in names. Here are multiple-word examples of each type:

TSortedList (class name)

fSubViews (data member of class)

DrawContents (function member of class)

gDeviceList (global or static data member)

theCurrentSize (local or parameter)

kMaxStringLength (constant)

Names with global scope

Any name with global scope (for example, class names, typedefs, constants, globals) should have a distinctive and unique name. This will help avoid name conflicts. Names like Short and Number are fairly nondescript and likely to wind up conflicting with identifiers from other header files accidentally (this is a big problem with MPW and the ROM interfaces today). Better in these cases would be kShortLived (to follow our advice on constant names) or StringLength (more descriptive of the function). When you name something with global scope, think about the fact that it's in a global name space and someone may have to figure out what it is without context. Use more specific names rather than more general ones.

C++ relieves this problem somewhat by adding enumerations with class scope and static members. Enumerations declared inside classes are accessible using qualification, as in

```
class TFoo {
public:
   enum {kFred, kBarney};
...
};

i = TFoo::kFred;
```

This lets you put constants associated with different classes into different name spaces, somewhat like the way C changed a few years back so that structure members from different `structs` were in different name spaces.

Static members let you put ordinary functions and global variables associated with a class into the scope of the class. For example,

```
class TView {
public:
   static void Initialize();
   static const TView kWhizzyView;
   static const long kMagicNumber;
...
};

TView::Initialize();
...TView::kWhizzyView...
i = TView::kMagicNumber;
```

Putting such global functions and variables into the scope of the class helps avoid name collisions. In fact, we frown on the use of ordinary globals: most global functions and variables should be static members of some class. The same with constants: they should be made members of an enumeration inside a class, if possible. Of course, global variables that are not constants of the sort shown above shouldn't be public at all; instead, access should be through member functions, static or normal:

```
class TFoo {
public:
   static Boolean gWhoopeeFlag;  // BAD!
}

TFoo::gWhoopeeFlag = TRUE;     // BAD!
```

The preprocessor

One of the most powerful features of the C and C++ languages is the powerful C preprocessor.

Don't use it.

Except for include files and conditional compilation, C++ has features that supersede most of the techniques that used the preprocessor. Sometimes you need to use the preprocessor to accomplish things you can't with C++, but far less often than with straight C.

Use `const` for constants

Don't use `#define` for symbolic constants. Instead, use C++'s `const` storage class. As with `#define` symbols, these are evaluated at compile time. *Unlike* `#define` symbols, however, they follow the C scope rules and have types associated with them. You can also use `enum`s. For example:

```
#define kGreen 1 // No no
const int kGreen = 1; // Better
enum Color {kRed, kGreen, kBlue} // Best
```

This prevents a host of problems. With `#define` symbols, for example, if you accidentally redefine a name, the compiler will silently change the meaning of your program. With `const` or `enum`s, you'll get an error message. Even better, with `enum`s you can put the identifiers in the scope of an enclosing class: see "Naming Conventions," above. As an added bonus, each enumeration defined is treated as a separate type for purposes of type checking (much as scalars are handled in Pascal).

Unlike ANSI C, C++ considers objects that are declared `const` and initialized with compile-time expressions to be compile-time constants (but only if they are of integral type). Thus, they can be used as case labels, for example.

Use `enum` for a set of constants

If your constants define a related set, don't use separate const definitions. Instead, make your constants an enumerated type. For example:

```
// Bleah.
const int kRed = 0;
const int kBlue = 1;
const in kGreen = 2;

// Allll Riiiight!
typedef enum {kRed, kBlue, kGreen} ColorComponent;
```

This causes ColorComponent to become a distinct type that is type-checked by the compiler. Values of type ColorComponent will be automatically converted to int as needed, but integers cannot be changed to ColorComponents without a cast. If you need to assign particular numerical values, you can do that too:

```
typedef enum {kRed = 0x10, kGreen = 0x20, kBlue = 0x40} ColorComponent;
```

Where possible, the type declaration should occur within the scope of a class. Then, references to the constants outside of the class's member functions must be qualified:

```
class TColor {
public:
   typedef enum {kRed, kGreen, kBlue} ColorComponent;
...
}

foo = TColor::kRed;
```

Note that the enum type name is not local to the class; only the actual constants. The enum type name should not be qualified.

Use `inline` for macro functions

Function macros are another source of fun problems in C programs, like this classic example:

```
#define SQUARE(x) ((x)*(x))
SQUARE(y++);
```

C++ allows functions to be declared inline (see also below), which completely obviates the need for function macros. Like const, inline functions follow the C++ scope rules and allow argument type checking. Both member functions and nonmember functions can be declared inline. So the above example becomes

```
inline int Square(int x)
{
   return x*x;
};
Square(y++);
```

This code does the right thing, and is actually more efficient than the macro version (as well as being correct).

Okay for preprocessor control

As stated above, the preprocessor is necessary for `#include` files, and preprocessor symbols are necessary for conditional compilation.

Use of `const`

Both ANSI C and C++ add a new modifier to declarations, `const`. This modifier lets you declare that the specified object cannot be changed. This lets the compiler optimize code, and also warn you if you do something that doesn't match the declaration. Here are some example of const declarations:

```
const int *foo;
```

This is a modifiable pointer to constant integers. foo may be changed, but what it points to may not be.

```
int *const foo;
```

This is a constant pointer to modifiable integers. The pointer cannot be changed (once initialized), but the int it points to can be changed at will.

```
const int *const foo;
```

This is a constant pointer to a constant int. Neither the pointer nor the int it points at may be changed.

Note that const objects can be assigned to non-const objects (thereby making a copy), and the modifiable copy can of course be changed. However, *pointers* to const objects **may not** be assigned to pointers to non-const objects, although the converse is allowed. Both of these assignments are legal:

```
(const int *) = (int *);
(int *) = (int *const);
```

Both of these assignments are illegal:

```
(int *) = (const int *);
(int *const) = (int *);
```

All of these rules apply to class objects as well; you can declare something `(const TView *)`. There used to be a hole in the language, however: you could call any member function of an object using a const pointer to it, and that member function could modify the object (since there was no way to declare which member functions modify the object). For example, this was legal:

```
const TView *aView;
...
aView->ModifySomething();
```

To plug this hole, member functions that will be called for const objects must now be declared const; see the 1985–1989 paper for details. The syntax looks like this:

```
class TFoo {
public:
   void Bar1() const;
   void Bar2();
};
...
const TFoo *fp;
fp->Bar1(); // legal
fp->Bar2(); // illegal (actually, just a warning for now)
```

Note that inside a `const` member function, the `this` pointer has type `const TFoo *`, so you really can't change the object. You could cast the pointer to be just a `TFoo *`, but then you may be surprising your clients. Even though you think that your change to the object is not externally visible (that is, it doesn't change the state of the object as far as clients are concerned—one example is an internal cache), it can have an impact. If your object is being used by an interrupt routine that "reads" it, your client may assume that it's okay to call a `const` member function, since he or she thinks the object isn't going to change. However, if the internal state of the object changes anyway, access by multiple "readers" will cause its state to become corrupted.

Another example is an object placed in ROM. The client thinks it's all right to call a `const` member function of the object, and then gets a bus error because the attempted write access fails.

The bottom line is that if you attempt to cast your `this` pointer to a non-`const` version inside a `const` member function, you had better think through the implications of this for your clients in an interrupt-driven environment, and you had better document it.

Use of language features

In this part you'll find advice on using particular features of the C++ language. The topics are arranged roughly in order of increasing difficulty.

Global variables

Because they can be accessed by any function in a program, global variables are risky to use. C provides alternative ways of sharing data between functions, and C++ provides one more.

Static class members do the job of global variables

Static class members do have one major advantage over regular globals: scope. Regular globals (that is, static external variables) have global scope. That means there are potential name collisions with globals from any other include file the developer may use. Static class members, however, have full scoping: they're qualified by the name of their class, and don't conflict with any identifier outside the class. They can also be protected. If you were going to have a simple global, consider a static member instead.

Be careful about static initialization

If you design a class that depends on some other facility in its constructor, be careful about order dependencies in static initialization. The order in which static constructors (that is, the constructors of objects with static storage class) get called is **undefined.** You cannot count on one object being initialized before another. Therefore, if you have such a dependency, you must either document that your class cannot be used for static objects, or you must use "lazy evaluation" to defer the dependency until later.

Inline functions

Now that we've told you about inline functions, never use them. Well, hardly ever. The main reason is that they get compiled into your caller's code. This makes them a tad difficult to override. Also, you have to ship their source code to everyone. There are a few times when it's okay to use them:

Okay to use if it expands to call to something else

If your inline function just calls something else that isn't inline, that's fine, as long as the other function **has identical semantics.** An example: You might have a class that defines a virtual function IsEqual, which compares two objects for equality. It also has an inline definition for the operator ==, as a notational convenience. Since operator == just turns around and calls the IsEqual function, it's okay for it to be inline and not virtual. This does **not** apply if your function just happens to have a one-line implementation.

Inline functions sometimes speed your program

Of course, the other time it's okay to use `inline` is if efficiency is extremely important. Note that code size may increase due to duplication of code. **You may actually decrease system performance by making something inline,** since you're increasing the amount of code that must fit in RAM. Also, once a function is more than a couple of lines in length, the function call overhead is a very small fraction of the total time, and you are not buying much by making it inline.

An example is addition for a complex number type; here, the efficiency consideration together with the low probability of a future change makes an inline implementation a good idea. Also, we're not exactly talking trade secret with regard to shipping the source. The complex number implementation shipped with C++ makes addition and subtraction inlines (fairly short), but makes multiplication and division regular functions (they are longer, so the overhead for a call is less important and the code size issue is more important).

If you don't **know** that your implementation must be inline, **don't make it inline.** Build it normally and then **measure** the performance. Experience has shown again and again that programmers spend lots of time optimizing code that hardly ever gets executed, while totally missing the real bottlenecks. The empirical approach is much more reliable. Experience has also shown that a better algorithm or smarter data structures will buy you a lot more performance than code tweaking.

Don't write inlines in declarations

C++ has two ways of declaring an inline member function. One is to declare the member function normally and then supply an inline function definition later in the same header file. The other is to write the function definition directly in the class declaration. Never use this latter form; always declare the function normally and then put an inline definition at the end of the file. That way, it's much easier to change between inline and regular implementations of a function, and it's no less efficient. The fact that something is inline should not be made obvious in the class declaration, since clients may start counting on it.

```
class TFoo {
public:
   int TweedleDee() { return 1; };  // Bad!
   int TweedleDum();           // Good!
};

inline int TFoo::TweedleDum()
{
   return 2;
}
```

Unspecified arguments

It's possible to partially circumvent the strong type checking C++ imposes on function arguments. You should avoid doing this if at all possible.

Don't use unspecified arguments

C++, like ANSI C, allows you to cling to C's past by declaring functions that take unspecified numbers and types of arguments, the classic example being

```
void printf(char *, ...);
```

Very few functions need to have an interface like this. If you want to be able to omit arguments, for example, you can use default arguments or function overloading (both defined below).

Do use default arguments, but cautiously

A better technique than unspecified arguments is default arguments. You can specify default values for arguments that are only used sometimes. This is especially handy in constructors. For example:

```
TView::TView(TVPoint itsSize, TVPoint itsLocation,
       TView *itsSuperView = NIL);
```

This function can be called either with three arguments or with two. This can help you avoid that agonizing decision as to whether to include an option or not. However, be sparing; long strings of defaults can make it hard to figure out what's going on. Further, you can only leave off arguments on the end, not in the middle, so if you have 10 defaults and someone wants to specify the last one, they must specify the preceding nine as well. This sort of defeats the idea. More than two default arguments is a bad idea, and even two is questionable.

Also remember that for both default arguments and function overloading, too many versions of the same function decrease the safety provided by type checking, and make it more likely that you will accidentally call a different version than the one you intended.

Function name overloading

C++ also lets you overload function names, by letting two functions (member or non-member) have the same name as long as the types of their arguments differ. This can be very useful but also can cause problems.

Implications (including errors)

This feature is useful when you want to have different versions of the same function; they should all be related. For example, you may want to have a constructor that takes lots of options, as well as one that is simple to use. Also, you may want to make functions that take different types. Examples include:

```
Rectangle::Rectangle(Point leftTop, Point rightBottom)
Rectangle::Rectangle(short top, short left, short bottom, short right)
void TPort::MoveTo(short, short)
void TPort::MoveTo(Point)
TComplex TComplex::Add(TComplex)
TComplex TComplex::Add(int)
```

The biggest problem is the unintentional use of the wrong argument type when overloading, which defines a new function. If TView has a member function

```
TView::Print(const TPrintRecord *)
```

and you define a subclass where, intending to override this function, you declare

```
TMyView::Print(const TStdPrintRecord *)
```

or

```
TMyView::Print(TPrintRecord *)
```

or even

```
TMyView::print(const TPrintRecord *)
```

because you forgot the type of the original argument (or misspelled the name), C++ assumes you want to overload the function and simply declares it as a new function. You have **not** overridden the original; it's still available. The latest version of CFront will warn you about the first two cases, but not the last.

Also remember that, as mentioned above, the more variants there are of a function, the easier it is to call the wrong one unintentionally because the arguments you supply just happen to match another variant.

Interaction with overriding of member functions (virtual or otherwise)

If you have an overloaded member function (whether virtual or not), and you override it in a derived class, then your override hides all overloaded variants of that member function, **not just the one you overrode.** Thus, if you want to override an overloaded member function, you must override **all** of the overloaded variants. C++ treats the overloaded function as a single entity; the scope resolution rule for C++ is to find the first class that has any function with that name defined, then look for a match based on argument type. The C++ team at AT&T believes this is the correct rule; their reasoning is that an overloaded set of functions is really just one function with a bunch of variants, and that you should not be naming functions with the same name unless they are really the same function.

An example that illustrates this behavior follows:

```
class A {
public:
   void Foo(long);
   void Foo(double);
};

class B: public A {
public:
   void Foo(double);
};

B bar;
bar.Foo(2);
```

The call actually winds up calling B::Foo(double) after coercing the integer argument to double.

On a positive note, CFront will warn you if you override some but not all of a set of overloaded member functions. See the latest 1985–1989 paper and the reference manual for details.

Be careful, as with operator overloading

Like operator overloading, function overloading can be abused. Functions should not have the same name unless they basically perform the same operation, as in the examples above. If that is the case, then having the functions be identically named can be a great help in reducing the number of things that a programmer must remember.

Operator overloading

Another C++ feature is the ability to define operators for your own classes. If you define a fixed-point data type, C++ lets you define the standard arithmetic operators for it, which makes code a lot easier to read.

Use only where appropriate and clear

Operator overloading also has tremendous potential for abuse. Defining the + operator for fixed-point numbers helps clarify code. Defining it as set union is also fairly clear. Defining =>> to mean "send a message" is crazy. Operator redefinition only helps when the new function is similar to the standard meaning of the operator; otherwise, it just confuses people. An example is C++'s streams facility, which redefines << and >> as output and input operators. This confuses a lot of people.

How do you call a base class's operator?

C++ occasionally delivers a pleasant surprise. One example is the syntax for calling overloaded operators. Of course, you can use the usual inline operator syntax; that's why C++ has operator overloading. The surprise is that you can also use functional syntax, which is sometimes essential, especially for calling a base class's operator. Here is an example of a subclass operator using the base class's operator:

```
const TWindow& TWindow::operator=(const TView &v)
{
   this->TView::operator=(v);
   return *this;
}
```

(The explicit "this->" is actually unnecessary here but was included for clarity.) In this example, TWindow has a base class TView, and we want to be able to assign a TView to a TWindow by just copying the TView part and leaving the rest of TWindow alone. To do this, we want to use TView's assignment operator. The functional notation here is the only way to do it. The pleasant surprise was that this notation is allowed.

Type coercion.

When to use type coercion

Like so many C++ features, type coercion can either clarify or obfuscate your code. If a type coercion seems "natural," like the coercion between reals and integers, then providing a coercion function seems like a good idea. If the conversion is unusual or nonsensical, then the existence of a coercion function can make it very hard to figure out what's going on. In the latter case, you should define a conversion function that must be called explicitly.

In general, coercion operators are useful in a way similar to operator overloading, and the same guidelines make sense.

Define type coercion rules for C++ to use

C++ will automatically coerce one type to another, but only if there is a direct way of doing so. In other words, if there is a conversion defined from type A to type B, C++ will use it automatically where appropriate. It will *not* concatenate coercion operators if there is not a direct coercion. So even though there may be a conversion defined from type A to type B, and type B to type C, C++ will not automatically convert type A to type C (you can do it explicitly via casts, though).

There are two ways of defining type coercions for C++ to use: constructors and type coercion operators. They are appropriate under different circumstances. Note that since all coercion operators must be member functions of some class, it is not possible to define a coercion from one nonclass type to another nonclass type. Also note that if more than one function is defined to perform the same coercion, they cannot be used implicitly or via a cast, since there is an ambiguity as to which to call. They can still be invoked explicitly.

Using a constructor

If you have a constructor with a prototype that looks like any of the following:

```
TargetClass::TargetClass(SourceType)
TargetClass::TargetClass(SourceType &)
TargetClass::TargetClass(const SourceType &)
```

then C++ will use it to convert from SourceType to TargetClass where appropriate. This form is useful when (1) the target type is a class (it can't be used for a primitive target type) and (2) the author of the target type wants to provide a coercion. If either of these conditions doesn't hold, you can use the second form of type coercion.

Using a type coercion operator

If the source type is a class, you can define a member operator function to perform the coercion. These operators have prototypes that look like:

```
SourceClass::operator TargetType()
```

TargetType may be either a primitive type or a class. It need not be the name of a type; it can be any type specifier (as long as it does not contain "array of" `[]` or "function" `()` forms; those must be handled via a typedef). This form is appropriate when the target type is not a class, or the source code for the target type is not available (that is, the coercion is being provided by someone else).

Encapsulation and data hiding

C++ provides several mechanisms to help you restrict access to data and prevent unwanted side effects. These guidelines will help you use these mechanisms.

Explicit use of `public`, `private`, `protected`

C++ thoughtfully allows you to leave out the `private` keyword in several places. Don't: it decreases C++'s well-known clarity. Class definitions should always explicitly state the visibility of their members and base classes. Write it like this:

```
class TFoo: public TBar, private MBaz {

public:
   public members;

protected:
   protected members;

private:
   private members;
};
```

Your `public` interface should come before your `protected` interface, and since your `private` interface is only necessary to make the compiler happy, it should be last.

No public or protected members that aren't functions

Always make *all* member variables private. (It's okay to make functions protected or public.) You can provide access functions to get and set your variables if you want (although you should think about exporting a more abstract interface instead). If you're really concerned with performance, you can make those functions inline (but see below on inline functions). Remember, don't compromise for the sake of performance **until you have numbers** to base your decision on!

What does `protected` really mean?

What the `protected` access mode means is not completely clear from Bjarne's various books and papers, so we will attempt to clarify the issue.

When a member of a class is declared protected, to clients of the class it is as if the member were private. In addition, however, subclasses can access the member as if it were declared private to them. This means that a subclass can access the member, but only as one of its own private fields. Specifically, it **cannot** access a protected field of its parent class via a pointer to the parent class, only via a pointer to itself (or a descendant). Here are some examples:

```
class A {
protected:
   void Bar();
};

class B: public A {
   void Foo();
};

class C: public B {
...
};

void B::Foo()
{
   A *pa;
   B *pb;
   C *pc;
```

```
    pa->Bar();      // Illegal:  A::Bar() is "private"
    Bar();             // OK:  "this" is of type B*
    pb->Bar();      // Also OK
    pc->Bar();      // Also OK
};
```

Protected constructors for abstract base classes

It's frequently useful to have a class that is not meant to be instantiated as an actual object, but only to be used as a base class for other classes. Examples include classes such as TApplication and TView. Such classes are called *abstract base classes.* If you want to enforce this status, you can make it impossible to instantiate such a class by making all of its constructors protected. In that case, the object cannot be created by itself, but only as part of a derived class.

In addition, there is a way to declare a member function abstract, that is, so that it must be overridden in descendants; this is called a *pure virtual function.* A class with such a member function cannot be instantiated, nor can any descendant class, unless all such functions are overridden. The syntax for this is as follows:

```
class Foo {
public:
    virtual void Bar() = 0;
};
```

You can also declare some (but not all) of the constructors for a class protected if you want those constructors to be used only by derived classes. The class can still be instantiated using the constructors that are public.

Private base classes

When you declare a base class `private` in C++, the derived class inherits all its members as `private` members. This means that the derived class is *not* considered a sub*type* of the base class, even though it is a sub*class*. You cannot pass a pointer to an object of the derived class when a pointer to the base is expected. This lets you inherit the behavior of a class without inheriting its type signature.

Since the derived class is not a subtype, it doesn't have an "is-a" relationship with the base class. Why not just make it a member, then? This is what you would normally do. However, if you need to reexport most of the functionality of the base class, you would have to write wrapper functions in your derived class that turned around and called the member class functions. Instead, you can use this devious shortcut.

By making the class a private base class, you don't inherit the type signature, but you do inherit the functionality, which can be made selectively visible without having to write wrapper functions. If A is a private base class of B, and B wants to make `A::Foo()` visible, then write the following in the declaration of B:

```
...
public:
   void A::Foo();
```

This code makes `Foo()` visible to clients of B.

Friends frowned upon but sometimes okay

Friend classes and functions are another C++ feature. Needless to say, they are a breach in the safety of types and in the integrity of the data abstraction provided by classes. If you have friends, you're probably doing something wrong.

About the only time this feature should be used is when implementing binary operators that can't be member functions. Another circumstance is a set of tightly related classes (an example from Bjarne's book is matrices and vectors). Generally, however, avoid friend classes and functions.

Hiding implementation classes

Sometimes a public class (one that you export to clients) must refer to a class that is only used in your implementation. How do you avoid exposing the implementation class? If the only reference is a pointer, then you can declare the implementation class as an incomplete class:

```
class TImplementation;

class TInterface {
...
private:
   TImplementation *fHidden;
};
```

This also works if your member functions have argument of type `TImplementation *` or `TImplementation &`. If you have actual TImplementation objects as fields, though, you must include the full declaration of TImplementation.

Don't expose yourself

The most important thing to remember is not to expose your implementation to either your clients or your subclasses (which are really just another kind of client). If you do so, you are tying the hands of those who must enhance your code.

It is very important to make sure that your class acts like a black box. The interface you export to clients and subclasses should reflect precisely what they need to know and nothing more. You should ask yourself, for every member function you export (remember, you're not exporting any data members, right?), "Does my client (or subclass) really need to know this, or could I recast the interface to reveal less?"

If you find that the interface to your class consists mostly of functions to get and set your private data members, you should ask yourself whether your object is really defining an abstract enough interface. The key is to think about the abstraction that your object represents and how clients view and use that abstraction, not how it is implemented. This is possibly *the* hardest thing to do in object-oriented design, but it is also one of the biggest advantages and has the biggest long-term payoff.

Read Alan Snyder's paper

For an excellent discussion of the issues involved in data abstraction, encapsulation, and typing, see Alan Snyder's paper "Encapsulation and Inheritance in Object-Oriented Programming Languages" in the 1986 OOPSLA proceedings.

Virtual functions

(Almost) all member functions should be virtual

Virtual functions are pretty inexpensive. Because of this, any class that is intended to be used in a polymorphic fashion (that is, a pointer to a subclass may be passed where a pointer to the class is expected) should have all of its functions virtual. It's hard to guess in advance which functions may be overridden in the future (although private functions can't be and so need not be virtual).

You should only use nonvirtual functions where you are very sure that the class (or this particular aspect of it) will **never have a subclass.** An example is a fixed-point data type, which is self-contained, or a graphics point, or other similar classes.

The assignment operator is also a special case. Assignment isn't inherited like other operators. If you do not define an assignment operator, one is automatically defined for you; it consists of calls to the assignment operators of all of your base classes and members (this is discussed in Stroustrup's "1985–89" paper). It is okay to make your assignment operator virtual, but it's only useful under rather specialized circumstances. A virtual function call will be generated for a virtual assignment operator only when the left-hand side of an assignment is a reference or a dereferenced pointer.

(Almost) all destructors should be virtual

What is a virtual destructor? This is actually something you should never have had to worry about. What do you think happens here?

```
class A {
   ~A();
};

class B: A {
   ~B();
};

A *foo = new B;
delete foo;
```

B::~B gets called, then A::~A, right? Wrong! Only A::~A gets called! In order for the right thing to happen, you must declare your destructors virtual, for example,

```
virtual ~A();
virtual ~B();
```

If you do this, the right destructors will get called. Needless to say, **any** class that has a virtual member function, inherits one, or is used in a polymorphic fashion **must** have its destructor declared virtual or problems will occur.

Use of virtual functions in constructors and destructors

If you call a virtual function in a constructor, be aware that the version of the function that corresponds to the constructor will be called, **not** the version that would normally be called. For example, if A has a method `foo`, B is a subclass of A, and B overrides `foo`, a call to `foo` from A's constructor calls `A::foo`, not `B::foo`. A call to `foo` from B's constructor does call B::foo. This is sufficiently confusing that it is best not to call a virtual function from a constructor at all. Naturally, this only applies to virtual functions of the object whose constructor is running; virtual functions of other objects (including those of the same class) are perfectly fine (unless they in turn call one of your virtual functions).

If you think about it, it does not make sense to call virtual functions from constructors and destructors. Since base class constructors are called before derived class constructors, and base class destructors are called after derived class destructors, the object is in a partially valid state. If a virtual function overridden in a derived class is called from the base class constructor, it may access derived class features that have not been initialized. Similarly, if it is called from the destructor, it may access features that have already been destroyed.

How to use them

For those of you who are coming from the non-object-oriented programming world, a word about use of virtual functions might be in order. Virtual functions should be used whenever you want to have more than one implementation of the same abstract class. They allow the system to defer the decision on which function to execute until run time.

The right way to use virtual functions is to structure them around well-defined abstractions. If someone is to override a virtual function, they must have a clear definition of what the function does, even if they only call the inherited version after a little bit of processing.

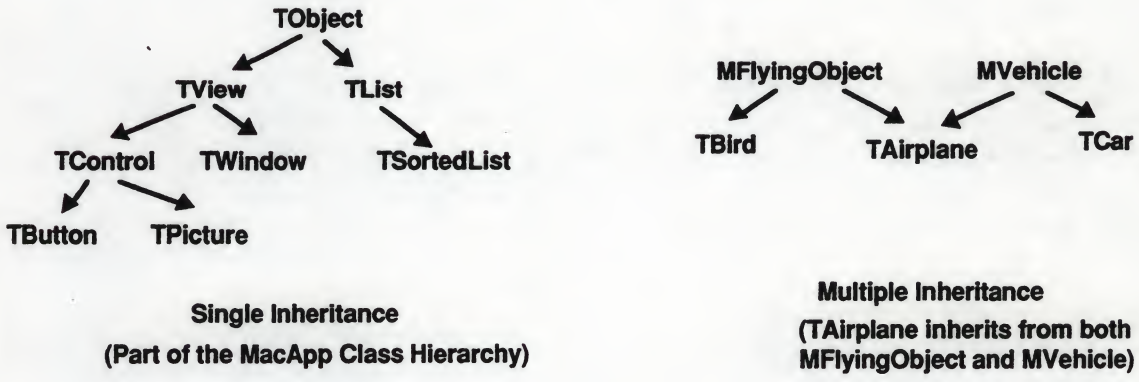
The **wrong** way to use virtual functions is via a "come-from" mechanism like so many Macintosh trap patches. Don't override a virtual function because "I know it's called from over here with these parameters." Needless to say, this wreaks havoc with the data abstractions, which are one of the major benefits of object-oriented programming. This is why the function must have a definition that is clear in terms of the object it belongs to, without any reference to its possible callers. If you override a function, the override must make sense in terms of the definition of the function itself.

Multiple inheritance

Multiple inheritance is a fairly new feature in object-oriented languages, and there is great potential for designing a confusing class hierarchy. The following guidelines explain how to avoid such chaos.

Figure E-1 is a diagram showing two class hierarchies. The one on the left shows a single-inheritance class hierarchy. Each class has only one parent. The diagram on the right shows a multiple-inheritance class hierarchy. Note that `TAirplane` inherits from both `MFlyingObject` and `MVehicle`.

- **Figure E-1** Single inheritance vs. multiple inheritance



Use in a controlled fashion

We start by artificially partitioning classes into two categories: base classes and mix-in classes. To distinguish the two, base class names begin with T (for example, TView), and mix-in class names begin with M (for example, MEditable). Base classes represent fundamental functional objects (like a car); mix-ins represent optional functionality (like power steering). The guidelines are as follows

- A class can inherit from *zero or one* base classes, plus *zero or more* mix-in classes. If a class does not inherit from a base class, it probably should be a mix-in class (though not always, especially if it is at the root of a hierarchy).
- A class that inherits from a base class is itself a base class: it cannot be a mix-in class. Mix-in classes can only inherit from other mix-in classes.

The net effect of these two rules is that the base classes form a conventional, tree-structured inheritance hierarchy rather than an arbitrary acyclic graph. This makes the base class hierarchy much easier to understand. Mix-ins then become add-in "options," which do not fundamentally alter the inheritance hierarchy.

Like all guidelines, this one is not meant to be hard and fast. Multiple inheritance can and should be used in other ways as well if it makes sense. The fundamental thing to keep in mind is that people (including programmers) are better at understanding regular structures than arbitrary directed acyclic graphs.

Why we should avoid virtual bases

As part of multiple inheritance, C++ contains a new feature called virtual base classes. If both B and C are subclasses of A, and D has both B and C as base classes, then D will have two A's if A is not virtual, but only one A if it is.

First, you shouldn't get into a situation like this if you follow the above guidelines for use of multiple inheritance. This is a very confusing situation, and no matter which alternative you choose programmers will have a hard time understanding it. Second, using virtual bases has a problem: once you have a pointer to a virtual base, there's no way to convert it back into a pointer to its enclosing class.

Multiple occurrences of a base

Sometimes the same base class (it should be a mix-in) will occur more than once as an ancestor of a class. It's only useful to have a base class twice if there are data members associated with it. It doesn't hurt to have a base class twice (aside from wasting space because of multiple pointers to the virtual function table), and if you need to cast back from the base class pointer to something else you may not have a choice. If you don't need to cast back, this is one of the situations where virtual base classes may be okay.

Design issues

This section discusses general problems of programming in C++.

Working in a value-based language

C++ has a different object model from Object Pascal or Smalltalk. The most fundamental difference is that whereas Object Pascal and Smalltalk are reference based (that is, like Lisp, assignment means copying a pointer), C++ is value based. By this we mean that classes in C++ are treated just like primitive types, whereas in Object Pascal objects are treated very differently from primitive types. This is actually a benefit, since all types in a program are handled in the same style, as opposed to the multiple styles in Object Pascal (Smalltalk, like C++, is also self-consistent). There are some implications for your C++ programming style, however.

Don't use pointers unless you mean it

Pointers should be used in C++ in the same way you would use them in plain C or plain Pascal; that is, when you really want multiple references to the same object, or a dynamic data structure. If you really just want to pass something by reference to avoid copying, then you can use a reference rather than a pointer (see below). In fact, you can even pass a class by value if the copying overhead isn't too high and you don't care about polymorphism (for example, when the class has no virtual functions).

Don't allocate storage unless you must

In a reference-based language like Object Pascal or Smalltalk, all objects must be heap allocated. In C++, it's better to treat values the same way you would in C. For example, instead of defining a Clone operator, overload the assignment operator; instead of allocating and returning an object, have the caller pass one in by reference and set it. This allows your classes to be treated just like primitive types, and in the same style. In general, leave storage allocation up to the class client.

By doing so, you can make use of one of C++'s unique features: the ability to have automatic and static objects, and objects as members of classes. No matter how clever or efficient a storage allocator we have, it can never be as fast as allocating an object on the stack, or as part of another object. If an object can be local to a function there is no storage allocation overhead. Many objects have very localized scope and do not need to be allocated on the heap.

There is one exception to the rule about allocating an object and returning a pointer: you must do this when the type of the returned object may vary. Anytime that a function must choose what type object to return, the function must allocate the object, not the caller.

It's still all right for the caller to allocate storage even when the type of the object being passed in may vary, since references, like pointers, can be used polymorphically (that is, you can specify a `TSubFoo&` to an argument of type `TFoo&`). The key question is whether the caller or the function must determine the type. In the former case, leave allocation to the client; in the latter, the function must allocate the object on the heap and return it.

Summary: Pretend everything is a primitive

In summary, you should design your classes so that using them is just like using a primitive type in C. This will let the client use them in a style that is "natural" for C. In cases where you wish to avoid copying, pass arguments by reference. Use pointers only when you want a truly dynamic data structure, or when polymorphism demands it (note that references allow for polymorphism also, since they are really just a different kind of pointer).

Pointers vs. references

Pointers and references may seem similar, but they have important differences. These are discussed below.

Differences

C++ provides two very similar mechanisms for passing references to entities. One is the familiar pointer from classic C; the other is a new concept, the reference. Pointers are declared as follows:

```
TFoo *fooPtr;
```

But references are declared like this:

```
TFoo &fooRef;
```

A pointer must be dereferenced to access what it points to, but a reference can be used as is, and acts as a synonym for the object it refers to, for both fetching and storing. This makes it similar to other highly refined mechanisms, such as Fortran's equivalence statement (or VAR parameters in Pascal). The entity to which a reference refers may only be set when the reference is created; in this respect it is somewhat like a `const` pointer that gets a `virtual *` put in front of it wherever it is used, and puts a `virtual &` in front of the expression from which it is initialized. Here are two examples:

```
void Bump(int *ip)
{
   *ip += 1;
}
Bump(&j);

void BumpR(int &i)
{
   i += 1;
}
BumpR(j);
```

There are certain circumstances where references are mandatory, for example in overloading the assignment operator. In other cases, either a pointer or a reference may be used. The question is, which should be used when?

References should be used when a parameter is to be passed "by reference,"s in Pascal. This means that the called function is going to forget about the argument as soon as it returns. A regular reference should be used if you are going to modify the argument (`TFoo &`), and a `const` reference should be used if you are not going to modify it but don't want the overhead of call by value (`const TFoo &`).

Pointers should be used when the function you are calling is going to retain a reference (an *alias*) to the object you are passing in, such as when you are constructing a dynamic data structure. An example is putting an object into a MacApp TList: the TList retains a pointer to your copy of the object. The explicit use of pointers lets the reader know that aliasing is occurring.

By using pointers and references appropriately, you can increase the readability of your code by giving the reader hints as to what is going on.

Portability

The following guidelines are intended to help you write more portable, and possibly more robust, code.

Don't make assumptions

You might someday want to run your code on a CRAY or a VAX™. Don't make assumptions that are only valid for the 680x0 family of processors. For example:

- Don't assume that `int` and `long` are the same size.
- Don't assume that an object of type `long`, `float`, `double`, or `long double` can be at any even address.
- Don't assume you know the memory layout of a data type.
- Especially don't assume you know how structures or classes are laid out in memory, or that they can be written to a data file as is.
- Don't assume pointers and integers are interchangeable. Use `void *` if you want an untyped pointer, not `char *`.
- Don't assume you know how the calling conventions are implemented, or indeed any detail of the language implementation or run time.

ANSI specifies the following about C's built-in types. **This is all you can safely assume:**

- `unsigned chars` can hold at least 0 to 255. They may hold more.
- `signed chars` can hold –127 to +127. They may hold more.
- `chars` may be either `unsigned` or `signed chars`. You can't assume either. Therefore, don't use `char` unless you don't care about sign extension.
- `shorts` can hold at least –32,767 to 32,767 (signed) or 0 to 65,535 (unsigned).
- `longs` can hold at least –2,147,483,647 to 2,147,483,647 (signed) or 0 to 4,294,967,295 (unsigned).
- `ints` can hold at least –32,767 to 32,767 (signed) or 0 to 65,535 (unsigned). In other words, **`ints` cannot be counted on to hold any more than a `short`.** `int` is an appropriate type to use if a `short` would be big enough but you would like to use the processor's "natural" word size to improve efficiency (on some machines, a 32-bit operation is more efficient than a 16-bit operation because you can avoid masking). **If you need something larger than a `short` can hold, you must specify `long`.**

If you need exact information, you can use the symbols defined in limits.h. or float.h. Remember, though, that the values of these symbols can change from processor to processor or compiler to compiler, within the limits defined above (for more information, see the ANSI C specification).

It's very easy to write nonportable code, and it takes some vigilance to avoid it. It's well worth the effort, however, the first time you port to a different processor, or try to use a different compiler.

Pick a canonical format for messages and data files

Remember that AppleTalk® networks connect to non-Apple computers such as the Intel-8x86 based MS-DOS machines. Thus, if you write or read any data in a context where it might go to or come from a different CPU you have to worry about formats. Such situations include reading or writing disk files, or sending data over a network (or even over NuBus®). The other CPU might even have a different byte order! The only solution to this problem is to pick a canonical format for your messages or data files.

Just because you have a canonical format doesn't mean you must pay a big overhead every time you access your data. An alternative is to perform the translation to or from canonical format at a predetermined time. For example, outline fonts (as defined by Royal) have a certain canonical format, which may not be convenient for a particular processor to deal with. However, they could certainly be converted to a convenient local format when the font is installed, rather than having to access them directly in their canonical format.

Don't use naked C types

Another way to make your life miserable is to use primitive C data types in your declarations. This is a bad idea, since if the implementation ever changes you have to do a lot of editing by hand. It's much better to declare a type (via class definition or `typedef`) that represents the abstract concept you want to represent, then phrase your declarations that way. This lets you change your implementation by simply editing the original type definition. Think of these types as giving your data physical units, like kilograms or meters/second. This prevents you from accidentally assigning a length to a variable with type Kilogram, catching more errors at compile time.

So instead of

```
long time;
short mouseX;
char *menuName;
```

use (for example)

```
typedef long TimeStamp;
typedef short Coordinate;
class TString { … };
...
TimeStamp time;
Coordinate mouseX;
TString menuName;
```

It's okay to use a raw C type under certain circumstances, such as when the quantity is machine dependent, or when it can be characterized as (for example) "a small integer." Otherwise, though, it's best to give yourself flexibility down the road.

Two ANSI C header files, "StdDef.h" and "Limits.h," contain useful definitions. Here are two of the more useful ones:

`size_t`	The type returned by the built-in C `sizeof` function. This is useful for representing the sizes of things.
`ptrdiff_t`	A type that can represent the difference between any two pointers.

The astute reader has noticed that these names do not conform to our guidelines. In the interest of clarity, however, we deem it better to use the names as defined by the ANSI C committee.

Another item worthy of note: if a data type is unsigned, declare it unsigned; this helps avoid problems later.

Use MacApp's `FAILURE` mechanism for error reporting

Returned error codes are (ironically) a very error-prone technique. MacApp's standard is to use exceptions: a structured technique for reporting errors back to a function's callers.

Unfortunately, the exception scheme does not handle an important case: an exception that occurs in a constructor. Handling this properly requires compiler support, since any base class or member constructor that has already been called must have its corresponding destructor called: only the compiler can know this. Until we get an official C++ scheme, you must handle this problem on a case-by-case basis.

Signaling an exception in a destructor is not a good idea, since any destructive behavior that has already taken place probably cannot be reversed.

Background reading

Here are some suggestions as to readings that may help you to use MPW C++ effectively.

Bring yourself up to date

You've just finished reading Bjarne's book, and you're feeling pretty smug. You've finally got C++ nailed.

Wrong.

You still have one more reference you must read. Read the paper "The Evolution of C++: 1985 to 1989". This paper is included with the AT&T C++ *Selected Readings* manual, which is part of the MPW C++ package.

At least one statement made in earlier versions of the paper (which were titled "The Evolution of C++: 1985 to 1987") is wrong. The order of execution of base class and member constructors and destructors is determined by their declaration order, not the order in which calls are made to such constructors in your constructor's header. This is a change to the language that was made after the 1985–87 paper was written; see the latest version (1985–89) for a full discussion.

Other books that give a good discussion of features that are new in C++ 2.0 are *The C++ Primer,* by Stanley Lippman, and *The C++ Answer Book,* by Tony Hansen. The latter book not only discusses 2.0 but also gives solutions to all of the problems in Stroustrup's book.

Read up on ANSI C

If you were whelped on good old K&R C, you may have a few surprises in store for you. There have been several changes to the language as part of the ANSI standardization process. If you learned C a while back, it might be a good idea to brush up on ANSI C. We highly recommend the second edition of Kernighan and Ritchie *(The C Programming Language),* which has appendixes that detail the differences between the original language and the ANSI version. Another good book is the second edition of Harbison and Steele *(C: A Reference Manual).* For purists, the ANSI C draft and rationale are available from ANSI itself.

Study object-oriented design

Doing a good job of object-oriented software design requires more than just learning an object-oriented language. The whole point of object-oriented languages is to permit a different approach to software design. This approach takes time and energy to learn. Without spending that time and energy, it's not possible to gain the full benefits of the approach.

That's why you should read *Abstraction and Specification in Program Development* and *Object-Oriented Software Construction.* The first book does not discuss object-oriented design per se, but it does cover the topic of data abstraction very well. Data abstraction is an important component of object-oriented design. The second book is a little pedantic in places, but has many, many good suggestions and ideas in it. Reading both will be hard work (especially since both are based on obscure languages), but will help you a great deal.

One example of an issue that *Object-Oriented Software Construction* covers quite well is the question of whether to use a class as a base (inherit from it) or a member (include it as a field). As Bertrand Meyer notes, the distinction is whether the new class can act as an instance of its base class (it **is an** object of that type) or uses an instance of the class (it **has an** object of that type). For example, an automobile **is a** vehicle, but it **has an** engine. For good discussions of this and other issues, read the book (this particular discussion starts on page 333).

Taking the time to learn how to design with objects may be painful (after all, we're all working as hard as we can already), but it can make a big difference in the quality of the system when it's done. Every extra minute you take to improve the design now will pay off in easier maintenance and enhancements later.

Index

P

R

S

T

U

V

W

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and Microsoft® Word software. Proof and final pages were created on Apple LaserWriter® printers. POSTSCRIPT®, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Apple's corporate font, a condensed version of Garamond. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

Writers: Don Reed, Meryle Sachs

(Appendix E, "MPW C++ Style Guide," was originally written by David Goldsmith and Jack Palevich and published in a somewhat different form in the April 1990 issue of *develop, the Apple Technical Journal.*)

Product Management: Harvey Alcabes, Tim Swihart

Engineering Management: Dave Burnard, Ken Friedenbach, Preston Gardner

Engineers: Bill Gibbons, David Goldsmith, Cliff Greyson, Keithen Hayenga, Herb Kanner, Jim Loftus, Steve Lurya, Joe MacDougald, Jack Palevich, Jeannette Robertson, Arn Schaeffer, Andy Shebanow

Special thanks to: Eagle Berns, Dave Bice, Fred Forsman, Jordan Mattson, Linda Suits